FUNDAMENTOS DE ALGORITMIA

G. Brassard / P. Bratley

Département d'informatique et de recherche opérationelle
Université de Montréal

Traducción:
Rafael García-Bermejo
Facultad de Físicas
Universidad de Salamanca

Revisión técnica:
Narciso Martí
Facultad de Matemáticas
Universidad Complutense de Madrid

Ricardo Peña
Escuela Superior de Informática
Universidad Complutense de Madrid

Luis Joyanes Aguilar
Facultad de Informática
Universidad Pontificia de Salamanca en Madrid

PRENTICE HALL
Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo
San Juan • San José • Santiago • São Paulo • White Plains

Contenido

PRÓLOGO xv

PRÓLOGO A LA EDICIÓN EN ESPAÑOL xxi

1. PRELIMINARES 1

1.1 INTRODUCCIÓN 1

1.2 ¿QUÉ ES UN ALGORITMO? 2

1.3 NOTACIÓN PARA LOS PROGRAMAS 7

1.4 NOTACIÓN MATEMÁTICA 8

1.4.1 Cálculo proposicional 8
1.4.2 Teoría de conjuntos 9
1.4.3 Enteros, reales e intervalos 10
1.4.4 Funciones y relaciones 11
1.4.5 Cuantificadores 11
1.4.6 Sumas y productos 13
1.4.7 Miscelánea 14

1.5 TÉCNICA DE DEMOSTRACIÓN 1: CONTRADICCIÓN 15

1.6 TÉCNICA DE DEMOSTRACIÓN 2: INDUCCIÓN MATEMÁTICA 18

1.6.1 El principio de inducción matemática 21
1.6.2 Un asunto completamente distinto 26
1.6.3 Inducción matemática generalizada 28
1.6.4 Inducción constructiva 31

1.7 RECORDATORIOS 35

1.7.1 Límites 35
1.7.2 Series sencillas 39
1.7.3 Combinatoria básica 44
1.7.4 Probabilidad elemental 48

1.8 PROBLEMAS 56

1.9 REFERENCIAS Y TEXTOS MÁS AVANZADOS 62

2 ALGORITMIA ELEMENTAL 65

2.1 Introducción 65
2.2 Problemas y ejemplares 66
2.3 Eficiencia de los algoritmos 67
2.4 Análisis de «caso medio» y de «caso peor» 70
2.5 ¿Qué es una operación elemental? 73
2.6 ¿Por qué hay que buscar la eficiencia? 76
2.7 Ejemplos 78
2.7.1 Cálculo de determinantes 78
2.7.2 Ordenación 79
2.7.3 Multiplicación de enteros muy grandes 80
2.7.4 Cálculo del máximo común divisor 82
2.7.5 Cálculo de la sucesión de Fibonacci 83
2.7.6 Transformada de Fourier 85
2.8 ¿Cuándo queda especificado un algoritmo? 85
2.9 Problemas 86
2.10 Referencias y textos más avanzados 89

3 NOTACIÓN ASINTÓTICA 91

3.1 Introducción 91
3.2 Una notación para "el orden de" 91
3.3 Otra notación asintótica 98
3.4 Notación asintótica condicional 101
3.5 Notación asintótica con varios parámetros 104
3.6 Operaciones sobre notación asintótica 105
3.7 Problemas 106
3.8 Referencias y textos más avanzados 110

4 ANÁLISIS DE ALGORITMOS 111

4.1 Introducción 111
4.2 Análisis de las estructuras de control 111
4.2.1 Secuencias 112
4.2.2 Bucles "para" (desde) 112
4.2.3 Llamadas recursivas 114
4.2.4 Bucles "mientras" y "repetir" 116
4.3 Uso de un barómetro 118

4.4 Ejemplos adicionales 120
4.5 Análisis del caso medio 126
4.6 Análisis amortizado 127
4.7 Resolución de recurrencias 132
4.7.1 Suposiciones inteligentes 132
4.7.2 Recurrencias homogéneas 135
4.7.3 Recurrencias no homogéneas 140
4.7.4 Cambios de variable 148
4.7.5 Transformaciones de intervalo 156
4.7.6 Recurrencias asintóticas 157
4.8 Problemas 160
4.9 Referencias y textos más avanzados 166

5 ESTRUCTURAS DE DATOS 167

5.1 Matrices (arrays), pilas y colas 167
5.2 Registros y punteros (apuntadores) 170
5.3 Listas 171
5.4 Grafos 173
5.5 Árboles 175
5.6 Tablas asociativas 181
5.7 Montículos (heaps) 184
5.8 Montículos binomiales 193
5.9 Estructuras de conjuntos disjuntos (partición) 198
5.10 Problemas 204
5.11 Referencias y textos más avanzados 209

6 ALGORITMOS VORACES 211

6.1 Dar la vuelta (1) 211
6.2 Características generales de los algoritmos voraces 213
6.3 Grafos: árboles de recubrimiento mínimo 215
6.3.1 Algoritmo de Kruskal 217
6.3.2 Algoritmo de Prim 220
6.4 Grafos: Caminos mínimos 223
6.5 El problema de la mochila (1) 227
6.6 Planificación 230
6.6.1 Minimización del tiempo del sistema 231

6.6.2 Planificación con plazo fijo 233
6.7 Problemas 242
6.8 Referencias y textos más avanzados 245

7 DIVIDE Y VENCERÁS 247

7.1 Introducción: multiplicación de enteros muy grandes 247
7.2 El caso general 251
7.3 Búsqueda binaria 255
7.4 Ordenación 257
7.4.1 Ordenación por fusión 258
7.4.2 Ordenación rápida (Quicksort) 260
7.5 Búsqueda de la mediana 266
7.6 Multiplicación de matrices 272
7.7 Exponenciación 274
7.8 Ensamblando todas la piezas: introducción a la criptografía 279
7.9 Problemas 282
7.10 Referencias y textos más avanzados 288

8 PROGRAMACIÓN DINÁMICA 291

8.1 Dos ejemplos sencillos 292
8.1.1 Cálculo del coeficiente binomial 292
8.1.2 El campeonato mundial 293
8.2 Devolver cambio (2) 295
8.3 El principio de optimalidad 298
8.4 El problema de la mochila (2) 299
8.5 Caminos mínimos 301
8.6 Multiplicación encadenada de matrices 304
8.7 Enfoques que aplican recursión 309
8.8 Funciones con memoria 311
8.9 Problemas 312
8.10 Referencias y textos más avanzados 317

9 EXPLORACIÓN DE GRAFOS 319

9.1 Grafos y Juegos: introducción 319
9.2 Recorrido de árboles 326

9.2.1 Preacondicionamiento 327
9.3 RECORRIDO EN PROFUNDIDAD: GRAFOS NO DIRIGIDOS 329
9.3.1 Puntos de articulación 332
9.4 RECORRIDO EN PROFUNDIDAD: GRAFOS DIRIGIDOS 334
9.4.1 Grafos acíclicos: ordenación topológica 336
9.5 RECORRIDO EN ANCHURA 337
9.6 VUELTA ATRÁS 342
9.6.1 El problema de la mochila (3) 343
9.6.2 El problema de las ocho reinas 345
9.6.3 El caso general 348
9.7 RAMIFICACIÓN Y PODA 348
9.7.1 El problema de la asignación 349
9.7.2 El problema de la mochila (4) 353
9.7.3 Consideraciones generales 354
9.8 EL PRINCIPIO DE MINIMAX 354
9.9 PROBLEMAS 357
9.10 REFERENCIAS Y TEXTOS MÁS AVANZADOS 363

10 ALGORITMOS PROBABILISTAS 365

10.1 INTRODUCCIÓN 365
10.2 PROBABILISTA NO IMPLICA INCIERTO 366
10.3 TIEMPO ESPERADO FRENTE A TIEMPO PROMEDIO 368
10.4 GENERACIÓN DE NÚMEROS PSEUDOALEATORIOS 369
10.5 ALGORITMOS PROBABILISTAS NUMÉRICOS 371
10.5.1 La aguja de Buffon 371
10.5.2 Integración numérica 375
10.5.3 Conteo probabilista 377
10.6 ALGORITMOS DE MONTE CARLO 379
10.6.1 Verificación de la multiplicación de matrices 380
10.6.2 Comprobación de primalidad 383
10.6.3 ¿Puede un número ser probablemente primo? 387
10.6.4 Amplificación de la ventaja estocástica 390
10.7 ALGORITMOS DE LAS VEGAS 393
10.7.1 El problema de las ocho reinas, segunda parte 396
10.7.2 Selección y ordenación probabilistas 400
10.7.3 Tablas de dispersión universales 402
10.7.4 Factorización de enteros muy grandes 404
10.8 PROBLEMAS 409
10.9 REFERENCIAS Y TEXTOS MÁS AVANZADOS 416

11 ALGORITMOS PARALELOS 419

11.1 Un modelo para la computación paralela 419

11.2 Técnicas básicas 422
11.2.1 Cómputo con un árbol binario completo 422
11.2.2 Duplicación de punteros 424

11.3 Trabajo y eficiencia 428

11.4 Dos ejemplos de teoría de grafos 431
11.4.1 Caminos mínimos 431
11.4.2 Componentes conexos 432

11.5 Evaluación de expresiones en paralelo 435

11.6 Redes de ordenación en paralelo 443
11.6.1 El principio cero-uno 445
11.6.2 Redes de fusión en paralelo 446
11.6.3 Redes de ordenación mejoradas 448

11.7 Ordenación en paralelo 449
11.7.1 Preliminares 450
11.7.2 La idea clave 450
11.7.3 El algoritmo 451
11.7.4 Un esbozo de los detalles 452

11.8 Comentarios ackrca de las p-ram EREW y CRCW 453

11.9 Cómputo distribuido 455

11.10 Problemas 457

11.11 Referencias y textos más avanzados 459

12 COMPLEJIDAD COMPUTACIONAL 461

12.1 Introducción: un ejemplo sencillo 462

12.2 Argumentos de la teoría de la información 462
12.2.1 La complejidad de la ordenación 466
12.2.2 La complejidad, al rescate de la algoritmia 470

12.3 Argumentos del adversario 472
12.3.1 Búsqueda del máximo en un vector 473
12.3.2 Comprobación de la conectividad de un grafo 474
12.3.3 La mediana, segunda parte 475

12.4 Reducciones lineales 476
12.4.1 Definiciones formales 480
12.4.2 Reducciones entre problemas matriciales 482
12.4.3 Reducciones entre problemas de caminos mínimos 487

12.5 Introducción a la NP-completitud 491

12.5.1 Las clases P y NP ... 491
12.5.2 Reducciones polinómicas ... 496
12.5.3 Problemas NP-completos ... 500
12.5.4 Algunas demostraciones de NP-completitud ... 504
12.5.5 Problemas NP-difíciles ... 507
12.5.6 Algoritmos no deterministas ... 508

12.6 UN ZOO DE CLASES DE COMPLEJIDAD ... 511

12.7 PROBLEMAS ... 515

12.8 REFERENCIAS Y TEXTOS MÁS AVANZADOS ... 522

13 ALGORITMOS HEURÍSTICOS Y APROXIMADOS ... 525

13.1 ALGORITMOS HEURÍSTICOS ... 526
13.1.1 Coloreado de un grafo ... 526
13.1.2 El viajante ... 528

13.2 ALGORITMOS APROXIMADOS ... 529
13.2.1 El viajante métrico ... 529
13.2.2 El problema de la mochila (5) ... 532
13.2.3 Llenado de cajas ... 534

13.3 PROBLEMAS DE APROXIMACIÓN CON DIFICUTAD NP ... 536
13.3.1 Problemas de aproximación con dificultad absoluta ... 538
13.3.2 Problemas de aproximación con dificultad relativa ... 539

13.4 LO MISMO, PERO DISTINTO ... 541

13.5 ENFOQUES DE APROXIMACIÓN ... 545
13.5.1 Llenado de cajas, segunda parte ... 546
13.5.2 El problema de la mochila (6) ... 547

13.6 PROBLEMAS ... 550

13.7 REFERENCIAS Y TEXTOS MÁS AVANZADOS ... 553

REFERENCIAS ... 555

NOTAS FINALES ... 569

ÍNDICE ANALÍTICO ... 571

Capítulo 1

Preliminares

1.1 INTRODUCCIÓN

En este libro hablaremos de algoritmos y de algoritmia. Este capítulo introductorio define en primer lugar lo que queremos decir con estas dos palabras. Ilustraremos esta discusión formal mostrando varias formas de realizar una sencilla multiplicación. También las tareas de todos los días poseen profundidades ocultas. Además aprovecharemos la oportunidad para explicar por qué pensamos que el estudio de los algoritmos es a la vez útil e interesante.

A continuación explicaremos la notación que vamos a utilizar a lo largo de todo el libro para describir algoritmos. El resto del capítulo consta en esencia de recordatorios de cosas que esperamos que el lector ya haya visto en otros lugares. Después de una breve revisión de la notación matemática estándar recordaremos dos técnicas de demostración útiles: la demostración por contradicción y la demostración por inducción matemática. A continuación enumeramos algunos resultados acerca de límites, sumas de series, combinatoria elemental y probabilidad.

Los lectores que ya estén familiarizados con estos temas deberían de leer las Secciones 1.2 y 1.3, y examinar brevemente el resto del capítulo, obviando aquel material que ya les resulte conocido. Es preciso prestar especial atención a la Sección 1.6.4.

Aquellas personas que tengan olvidadas sus matemáticas básicas y sus ciencias de la computación, deberían cuando menos leer los principales resultados que presentamos para refrescar su memoria. Nuestra presentación es sucinta e informal y no pretende ocupar el puesto de cursos de análisis elemental, de cálculo ni de programación. La mayoría de los resultados que damos se necesitarán posteriormente en el libro; a la inversa, en los capítulos posteriores intentaremos no utilizar resultados que vayan más allá de las bases que se han reunido en este capítulo.

1.2 ¿QUÉ ES UN ALGORITMO?

Un algoritmo, nombre que proviene del matemático persa del siglo IX al-Khowârizmî, es sencillamente un conjunto de reglas para efectuar algún cálculo, bien sea a mano o, más frecuentemente, en una máquina. En este libro nos preocupan fundamentalmente los algoritmos que van a ser utilizados en una computadora. Sin embargo, también podrían incluirse otros métodos sistemáticos para calcular un resultado; los métodos que aprendimos en la escuela para sumar, multiplicar y dividir números, son también algoritmos, por ejemplo. Muchos monaguillos ingleses, aburridos por pláticas poco interesantes, pasan el tiempo calculando la fecha de Pascua y empleando el algoritmo que se emplea en el Devocionario Anglicano. El algoritmo más famoso de la historia procede de un tiempo anterior al de los antiguos griegos: se trata del algoritmo de Euclides para calcular el máximo común divisor de dos enteros.

La ejecución de un algoritmo no debe de implicar, normalmente, ninguna decisión subjetiva, ni tampoco debe de hacer preciso el uso de la intuición ni de la creatividad. Por tanto se puede considerar que una receta de cocina es un algoritmo si describe precisamente la forma de preparar un cierto plato, proporcionándonos las cantidades exactas que deben de utilizarse y también instrucciones detalladas acerca del tiempo que debe de guisarse. Por otra parte, si se incluyen nociones vagas tales como «salpimentar a su gusto» o «guísese hasta que esté medio hecho» entonces no se podría llamar algoritmo.

Una aparente excepción a esta regla es que admitiremos como algoritmos unos procedimientos que efectúan elecciones aleatorias acerca de lo que hay que hacer en una situación dada. El Capítulo 10 en concreto trata de estos *algoritmos probabilistas*. Lo importante aquí es que el término «aleatorio» no quiere decir arbitrario; por el contrario, utilizamos valores seleccionados de tal manera que la probabilidad de seleccionar cada uno de los valores es conocida y está controlada. Una instrucción tal como «seleccionar un número entre 1 y 6» si no se da ningún detalle más, no es admisible en un algoritmo. Sin embargo, sería aceptable decir «seleccionar un número entre 1 y 6 de tal manera que todos los valores tengan la misma probabilidad de ser seleccionados». En este caso, cuando se ejecute el algoritmo manualmente, quizá decidamos obedecer esta instrucción tirando un dado sin cargar; en una computadora, podríamos implementarlo utilizando un generador de números pseudoaleatorios.

Cuando se utiliza un algoritmo para calcular la respuesta de un problema concreto, lo normal es suponer que las reglas nos darán, si se aplican correctamente, la respuesta correcta. Un conjunto de reglas que calcula que 23 veces 51 es 1.170 no suele ser es útil en una práctica general. Sin embargo en algunas circunstancias, estos *algoritmos aproximados* pueden resultar útiles. Si deseamos calcular la raíz cuadrada de 2, por ejemplo, ningún algoritmo nos podrá dar una respuesta exacta en notación decimal, por cuanto la representación de $\sqrt{2}$ es infinitamente larga y no se repite. En este caso, nos conformaremos con que el al-

goritmo nos pueda dar una respuesta que sea tan precisa como nosotros decidamos: 4 dígitos de precisión, o 10 dígitos o los que queramos.

Lo que es más importante, como veremos en el Capítulo 12, es que hay problemas para los cuales no se conocen algoritmos prácticos. Para tales problemas, la utilización de uno de los algoritmos disponibles para encontrar la respuesta exacta requerirá en la mayoría de los casos un tiempo excesivo: por ejemplo, algunos siglos. Cuando esto sucede, nos vemos obligados, si es que necesitamos disponer de alguna clase de solución al problema, a buscar un conjunto de reglas que creamos que nos van a dar una buena aproximación de la respuesta correcta, y que podremos ejecutar en un tiempo razonable. Si podemos demostrar que la respuesta computada mediante este conjunto de reglas no es excesivamente errónea, tanto mejor. En algunas ocasiones ni siquiera esto es posible, y solamente podremos fiarnos de nuestra buena suerte. Este tipo de procedimiento, basado fundamentalmente en el optimismo y frecuentemente con un apoyo teórico mínimo, se denomina un *algoritmo heurístico* o simplemente una *heurística*. Obsérvese una diferencia crucial entre los algoritmos aproximados y la heurística: con los primeros podemos especificar el error que estamos dispuestos a aceptar; con la segunda no podemos controlar el error, pero quizá seamos capaces de estimar su magnitud.

En los doce primeros capítulos de este libro, a no ser que el contexto indique claramente lo contrario, suponemos que un algoritmo es un conjunto de reglas para calcular la respuesta *correcta* a algún problema. Por otro lado el Capítulo 13 trata en su totalidad de algoritmos aproximados y de heurística.

Ahora se puede definir la Algoritmia simplemente como el estudio de los algoritmos. Cuando nos disponemos a resolver un problema, es posible que haya toda una gama de algoritmos disponibles. En este caso, es importante decidir cuál de ellos hay que utilizar. Dependiendo de nuestras prioridades y de los límites del equipo que esté disponible para nosotros, quizá necesitemos seleccionar el algoritmo que requiera menos tiempo, o el que utilice menos espacio, o el que sea más fácil de programar y así sucesivamente. La respuesta puede depender de muchos factores, tales como los números implicados, la forma en que se presenta el problema, o la velocidad y capacidad de almacenamiento del equipo de computación disponible. Quizás suceda que ninguno de los algoritmos disponibles sea totalmente adecuado, así que tendremos que diseñar un algoritmo nuevo por nuestros propios medios. La Algoritmia es la ciencia que nos permite evaluar el efecto de estos diferentes factores externos sobre los algoritmos disponibles, de tal modo que sea posible seleccionar el que más se ajuste a nuestras circunstancias particulares; también es la ciencia que nos indica la forma de diseñar un nuevo algoritmo para una tarea concreta.

Consideremos la aritmética elemental como ejemplo. Supongamos que tenemos que multiplicar dos enteros positivos utilizando nada más que papel y lápiz. Si nos han educado en Norteamérica, lo más probable es que se multiplique sucesivamente el multiplicando por cada una de las cifras del multiplicador, tomadas de derecha a izquierda, y que se escriban estos resultados intermedios

uno tras otro, desplazando cada línea un lugar a la izquierda, y que finalmente se sumen todas estas filas para obtener la respuesta. Por tanto para multiplicar 981 por 1.234 se construirá una disposición de números como la de la figura 1.1(a). Si, por otra parte, le han educado a uno en Inglaterra, es más probable que trabajemos de izquierda a derecha, dando lugar a la distribución que se muestra en la figura 1.1(b).

```
   981                    981
  1234                   1234
------                 ------
  3924                   981
 2943                   1962
1962                    2943
 981                    3924
------                 ------
1210554                1210554

  (a)                    (b)
```

Figura 1.1. Multiplicación (a) americana (b) inglesa

Estos dos algoritmos para la multiplicación son muy similares: son tan parecidos, de hecho, que nos referiremos a ellos como el algoritmo «clásico» de la multiplicación, sin preocuparnos por cuál de ellos queremos decir. En la figura 1.2 se ilustra un tercer algoritmo distinto para hacer lo mismo.

981	1.234	1.234
490	2.468	
245	4.936	4.936
122	9.872	
61	19.744	19.744
30	39.488	
15	78.976	78.976
7	157.952	157.952
3	315.904	315.904
1	631.808	631.808
		1.210.554

Figura 1.2. Multiplicación à la russe

Se escriben el multiplicando y el multiplicador uno junto a otro. Se hacen dos columnas, una debajo de cada operando, repitiendo la regla siguiente hasta que el número de la columna izquierda sea un 1: se divide el número de la columna de la izquierda por 2, ignorando los restos y se duplica el número de la columna de la derecha sumándolo consigo mismo. A continuación se tachan todas las filas en las cuales el número de la columna izquierda sea par, y finalmente se su-

man los números que quedan en la columna de la derecha. La figura ilustra la forma de multiplicar 981 por 1.234. La respuesta obtenida es:

$$1.234 + 4.936 + 19.744 + 78.976 + \ldots + 631.808 = 1.210.554.$$

Este algoritmo, que algunas veces se llama multiplicación *à la russe* se parece al que se emplea en el *hardware* de una computadora binaria. Tiene la ventaja de que no es preciso memorizar ninguna tabla de multiplicación. Lo único que hay que saber es sumar, y también dividir un número por dos. Aunque no es el algoritmo que suele enseñarse en la escuela, ofrece ciertamente un método alternativo de papel y lápiz para multiplicar dos enteros positivos.

En las figuras 1.3 y 1.4 se ilustra otro algoritmo distinto para multiplicar dos enteros positivos. Una vez más, ilustraremos el método multiplicando 981 por 1.234. Para este algoritmo, sin embargo, es necesario que el multiplicando y el multiplicador tengan el mismo número de cifras y además se necesita que este número sea una potencia de dos, tal como 1, 2, 4, 8, 16, etc. Esto se arregla fácilmente añadiendo ceros por la izquierda si es necesario: en nuestro ejemplo, añadimos nada más un cero a la izquierda del multiplicando, transformándolo en 0981, de tal manera que ambos operandos tengan cuatro cifras.

	Multiplicar		Desplazar	Resultado
i)	09	12	4	108····
ii)	09	34	2	306··
iii)	81	12	2	972··
iv)	81	34	0	2754
				1210554

Figura 1.3. Multiplicación de 0981 por 1.234 mediante divide y vencerás

Ahora para multiplicar 0981 por 1.234 multiplicamos primero la mitad izquierda del multiplicando por la mitad izquierda del multiplicador (12), y escribimos el resultado (108) desplazado hacia la izquierda tantas veces como cifras haya en el multiplicador: cuatro, en nuestro ejemplo. A continuación multiplicamos la mitad izquierda del multiplicando (09) por la mitad derecha del multiplicador (34), y escribimos el resultado (306) desplazado hacia la izquierda tantas veces como la mitad de las cifras que haya en el multiplicador: dos, en este caso. En tercer lugar multiplicamos la mitad derecha del multiplicando (81) por la mitad izquierda del multiplicador (12), y escribimos el resultado (972) desplazado también hacia la izquierda tantas veces como la mitad de las cifras que haya en el multiplicador, y en cuarto lugar multiplicamos la mitad derecha del multiplicando (81) por la mitad derecha del multiplicador (34) y escribimos el resultado (2.754), sin desplazarlo en absoluto. Por último sumamos los cuatro resultados intermedios según se muestra en la figura 1.3 para obtener la respuesta 1.210.554.

	Multiplicar		Desplazar	Resultado
i)	0	1	2	0··
ii)	0	2	1	0·
iii)	9	1	1	9·
iv)	9	2	0	18
				108

Figura 1.4 Multiplicación de 09 por 12 mediante divide y vencerás

Si se ha seguido el funcionamiento del algoritmo hasta el momento, se verá que hemos reducido la multiplicación de dos números de cuatro cifras a cuatro multiplicaciones de números de dos cifras (09 × 12, 09 × 34, 81 × 12 y 81 × 34), junto con un cierto número de desplazamientos y una suma final. El truco consiste en observar que cada una de estas multiplicaciones de números de dos cifras se puede efectuar exactamente de la misma manera salvo que cada multiplicación de números de dos cifras requiere cuatro multiplicaciones de números de una cifra, algunos desplazamientos y una suma. Por ejemplo la figura 1.4 muestra cómo multiplicar 09 × 12. Calculamos 0 × 1 = 0, desplazado hacia la izquierda dos veces; 0 × 2 = 0, desplazado hacia la izquierda una vez; 9 × 1 = 9, desplazado a la izquierda una vez, y 9 × 2 = 18, sin desplazar. Finalmente sumamos estos resultados intermedios para obtener la respuesta: 108. Utilizando estas ideas se podría operar de tal manera que las multiplicaciones solamente implicasen a operandos de una cifra. (Aunque hemos descrito la figura 1.3 antes que la figura 1.4 esto solamente era para simplificar la presentación. Por supuesto, tenemos que hacer primero las cuatro multiplicaciones de números de dos cifras, puesto que utilizamos los valores calculados así cuando se hace la multiplicación de los números de cuatro cifras.)

Este algoritmo tan poco habitual es un ejemplo de la técnica denominada «divide y vencerás», que estudiaremos en el Capítulo 7. Si le parece improbable que pueda ser más rápido que el algoritmo clásico, estará usted en lo cierto. Sin embargo, veremos en el Capítulo 7 que es posible reducir la multiplicación de dos números grandes a *tres*, y no cuatro, multiplicaciones de números cuyo tamaño es aproximadamente la mitad, junto con un cierto número de desplazamientos y sumas. (Si le estimulan los desafíos, intente averiguar la forma de hacerlo.) Con esta mejora el algoritmo de multiplicación por divide y vencerás se ejecuta más deprisa en una computadora que con los métodos anteriores siempre que los números que haya que multiplicar sean suficientemente grandes. (Se conocen métodos todavía más rápidos para operandos *muy* grandes.) No es totalmente necesario que la longitud de los operandos sea una potencia de dos, ni tampoco que tengan la misma longitud. El problema 1.6 muestra un caso en el cual el algoritmo pueda ser útil en la práctica, aun cuando los algoritmos sean relativamente pequeños, e incluso cuando se utilizan cuatro multiplicaciones en lugar de tres.

Lo importante de todos estos ejemplos es que, incluso en un aspecto tan sencillo como la aritmética elemental, pueden estar disponibles varios algoritmos para que nosotros efectuemos las operaciones requeridas. Uno puede resultar

atractivo por su familiaridad, otro por la naturaleza elemental de los cálculos intermedios implicados y un tercero por su velocidad en una máquina. Al hacer un estudio más formal de las propiedades de los algoritmos (al utilizar la algoritmia, en otras palabras) podemos tomar una decisión sabia acerca de la técnica que se debe utilizar en cualquier situación dada. Como veremos, una buena elección puede ahorrar a la vez tiempo y dinero; en algunos casos, puede ser la diferencia entre el éxito y el fracaso al resolver algún problema grande y difícil. La meta de nuestro libro es enseñar a tomar este tipo de decisiones.

1.3 NOTACIÓN PARA LOS PROGRAMAS

Es importante decidir la forma en que vamos a *describir* nuestros algoritmos. Si intentamos explicarlos en español, descubriremos rápidamente que los lenguajes naturales no están en absoluto adaptados para este tipo de cosas. Para evitar la confusión, en el futuro especificaremos nuestros algoritmos dando el correspondiente *programa*. Suponemos que el lector está familiarizado con al menos un lenguaje de programación bien estructurado, tal como Pascal. Sin embargo, no nos limitaremos estrictamente a ningún lenguaje de programación concreto: de esta manera, los aspectos esenciales de un algoritmo no resultarán oscurecidos por detalles de programación relativamente poco importantes, y realmente no importa cuál sea el lenguaje bien estructurado que prefiera el lector.

Hay algunos aspectos de nuestra notación para los programas que merecen especial atención. Utilizaremos frases en español en nuestros programas siempre que esto produzca sencillez y claridad. De manera similar, utilizaremos el lenguaje matemático, tal como el del álgebra y de la teoría de conjuntos, siempre que sea necesario (incluyendo símbolos tales como $\div$ y $\lfloor\ \rfloor$ que se presentan en la Sección 1.4.7). Como consecuencia, una sola «instrucción» de nuestros programas puede tener que traducirse a varias instrucciones, quizá a un bucle **mientras**, si es que el algoritmo tiene que implementarse en un lenguaje de programación convencional. Por tanto no se debe esperar ser capaces de ejecutar directamente los algoritmos que presentemos: siempre será preciso hacer el esfuerzo necesario para transcribirlos a un lenguaje de programación «real». Sin embargo, este enfoque es el que más se ajusta a nuestro objetivo primordial, que es presentar de la forma más clara posible los conceptos básicos que subyacen a nuestros algoritmos.

Para simplificar todavía más nuestros programas, omitiremos casi siempre las declaraciones de magnitudes escalares (enteras, reales o booleanas). En aquellos casos en que sea importante, tal como en las **funciones** y **procedimientos** recursivos, todas las variables que se utilizan se toman implícitamente como variables *locales*, a no ser que el contexto indique claramente lo contrario. En este mismo espíritu de simplificación se evita la proliferación de instituciones **begin** y **end** que invaden a los programas escritos en Pascal: el rango de instrucciones tales como **si**, **mientras** o **for**, así como otras declaraciones como **procedimiento**, **función** o **registro** se muestra sangrando las instrucciones en cuestión. La instrucción devolver marca la finalización dinámica de un **procedimiento** o de una **función**, y en este último caso **proporciona** además el valor de la función.

En los **procedimientos** y **funciones** no se declara el tipo de los parámetros, ni tampoco el tipo de los resultados proporcionados por una **función**, a no ser que tales declaraciones hagan que el algoritmo sea más fácil de entender. Los parámetros escalares se pasan por valor, lo que significa que son tratados como variables locales dentro del **procedimiento** o la **función**, a no ser que se declaren como parámetros **var**, en cuyo caso se pueden utilizar para proporcionar un valor al programa llamante. En contraste, los parámetros **tipo matriz** se pasan por referencia, lo cual significa que toda modificación efectuada dentro del **procedimiento** o **función** se verán reflejados en la matriz que realmente se pase en la instrucción que haga la llamada.

Por último, supondremos que el lector está familiarizado con los conceptos de recursividad, **registro** y puntero. Estos dos últimos se denotan exactamente como en Pascal, salvo por la omisión de **begin** y **end** en los **records.** En particular, los punteros se denotan con el símbolo «↑».

Para finalizar esta sección, véase un programa para multiplicar *à la russe*. Aquí el símbolo «÷» denota la división entera: cualquier posible resto se descarta. Compárese este programa con la descripción informal en español del mismo algoritmo en la Sección 1.2. ¿Cuál prefiere usted?

```
función rusa(m, n)
    resultado ← 0
    repetir
        si m es impar entonces resultado ← resultado + n
        m ← m ÷ 2
        n ← n + n
    hasta que m = 1
    devolver resultado
```

1.4 NOTACIÓN MATEMÁTICA

Esta sección revisa la notación matemática que utilizaremos a lo largo del libro. Nuestra revisión es sucinta, por cuanto suponemos al lector familiarizado con la mayor parte de ella. Sin embargo, recomendamos leerla al menos de forma rápida porque presentamos la mayoría de los símbolos que se van a utilizar, y algunos de ellos (tales como $[i..j]$, $\mathbin{\ddot{\forall}}$, $\mathbin{\ddot{\exists}}$, lg, $\lfloor x \rfloor$, ÷ y $\mathbb{R}^{\geq 0}$) no gozan de aceptación universal.

1.4.1 Cálculo proposicional

Existen dos «valores de verdad», *verdadero* y *falso*. Una *variable booleana* (o *proposicional*) solamente puede tomar uno de estos dos valores. Si p es una variable booleana, escribimos p *es verdadero*, o bien simplemente p , para indicar p = *verdadero*. Esto es generalizable a todas las expresiones arbitrarias cuyo valor sea booleano. Sean p y q dos variables booleanas. Su conjunción $p \wedge q$, o p y q es *verdadero* si y sólo si p y q son verdaderos. Su disyunción $p \vee q$, o p o q es *verdadero* si y sólo si al

menos uno de entre p o q es *verdadero*. (En particular, la disyunción de p y q es *verdadera* cuando tanto p como q son verdaderos.) La negación de p que se denota como $\neg p$ o «no p», es *verdadero* si y sólo si p es falso. Si la verdad de p implica la de q escribiremos $p \Rightarrow q$, que se lee si p entonces q. Si la verdad de p es equivalente a la de q, lo cual significa que son ambos o bien *verdadero* o bien *falso*, entonces escribimos $p \Leftrightarrow q$. Podemos construir fórmulas booleanas a partir de variables booleanas, constantes (*verdadero* y *falso*), conectivas ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) y paréntesis de la forma evidente.

1.4.2 Teoría de conjuntos

Aun cuando revisemos aquí los principales símbolos que se utilizan en la teoría de conjuntos, suponemos que el lector ya está familiarizado con la noción de conjunto. Por tanto en lo que sigue no se proporciona una definición formal. A todos los efectos prácticos, resulta suficiente pensar que un conjunto es una *colección no ordenada de elementos distintos*. Un conjunto se dice *finito* si contiene un número finito de elementos; en caso contrario el conjunto es *infinito*. Si X es un conjunto finito, $|X|$, la cardinalidad de X denota el número de elementos que hay en X. Si X es un conjunto infinito podemos escribir que la *cardinalidad* de X es infinita. El conjunto vacío, que se denota como ø, es el conjunto único cuya cardinalidad es 0.

La forma más sencilla de denotar un conjunto es rodear la enumeración de esos elementos entre llaves. Por ejemplo, {2,3,5,7}, denota el conjunto de números primos de una sola cifra. Cuando no puede surgir ninguna ambigüedad, se permite el uso de puntos suspensivos, tal como en «$\mathbb{N}$= {0,1,2,3, ...} es el conjunto de números naturales».

Si X es un conjunto, $x \in X$ significa que x *pertenece a* X. Escribiremos que $x \notin X$ cuando x *no pertenezca a* X. La barra vertical « | » se lee en la forma «tal que» y se utiliza para definir un conjunto describiendo la propiedad que cumplen todos sus miembros. Por ejemplo, $\{n \mid n \in \mathbb{N}$ y n es impar$\}$ denota el conjunto de todos los números naturales impares. Hay otras notaciones alternativas más sencillas para el mismo conjunto que son $\{n \in \mathbb{N} \mid n$ es impar$\}$ o incluso $\{2n + 1 \mid n \in \mathbb{N}\}$.

Si X e Y son dos conjuntos, $X \subseteq Y$ significa que todos los elemento de X pertenecen también a Y; y se lee «X es un *subconjunto* de Y». La notación $X \subset Y$ significa que $X \subseteq Y$ y además que hay por lo menos un elemento de Y que no pertenece a X; se lee «X es un subconjunto *propio* de Y». Tenga en cuenta que algunos autores utilizan $\subset$ para denotar lo que nosotros denotamos mediante $\subseteq$. Los conjuntos X e Y son iguales, lo cual se escribe $X = Y$, si y sólo si contienen exactamente los mismos elementos. Esto es equivalente a decir que $X \subseteq Y$ y que $Y \subseteq X$.

Si X e Y son dos conjuntos, denotamos su unión mediante $X \cup Y = \{z \mid z \in X$ o $z \in Y\}$, su *intersección* como $X \cap Y = \{z \mid z \in X$ y $z \in Y\}$, y su diferencia como

$$X \setminus Y = \{z \mid z \in X \text{ pero } z \notin Y\}.$$

Obsérvese en particular que $z \in X \cap Y$ cuando z pertenece tanto a X como a Y.

Representamos por (x, y) el *par ordenado* que consta de los elementos x e y en este orden. El *producto cartesiano* de X e Y es el conjunto de pares ordenados cuyo primer componente es elemento de X y cuyo segundo componente es elemento de Y; esto es $X \times Y = \{(x,y) \mid x \in X \text{ e } y \in Y\}$. Las n-tuplas ordenadas para $n > 2$ y el producto cartesiano de más de dos conjuntos se definen de forma similar. Denotaremos $X \times X$ por X^2 y similarmente para X^i, $i \geq 3$.

1.4.3 Enteros, reales e intervalos

Denotaremos el conjunto de los *números enteros* por $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$, y el conjunto de *números naturales* como $\mathbb{N} = \{0, 1, 2, ...\}$, y el conjunto de los *enteros positivos* como $\mathbb{N}^+ = \{1, 2, 3,...\}$. A veces ponemos de manifiesto que el 0 no está incluido en $\mathbb{N}^+$ haciendo alusión explícita al conjunto de los números enteros *estrictamente* positivos. En algunas ocasiones aludiremos a los números naturales con el nombre de enteros *no negativos*.

Indicamos el conjunto de *números reales* como $\mathbf{R}$, y el conjunto de los *números reales positivos* como

$$\mathbb{R}^+ = \{ x \in \mathbb{R} \mid x > 0\}$$

En algunas ocasiones hacemos hincapié en que 0 no está incluido en $\mathbb{R}^+$ aludiendo explícitamente al conjunto de números reales *estrictamente* positivos. El conjunto de números reales *no negativos* se denota mediante $\mathbb{R}^{\geq 0} = \{ x \in \mathbb{R} \mid x \geq 0\}$.

Un *intervalo* es un conjunto de números reales que yacen entre dos límites. Sean a y b dos números reales tales que $a \leq b$. El intervalo *abierto* (a, b) se representa por:

$$\{ x \in \mathbb{R} \mid a < x < b\}$$

El intervalo es vacío si $a = b$. El intervalo *cerrado* $[a,b]$ se representa por

$$\{ x \in \mathbb{R} \mid a \leq x \leq b\}$$

También existen intervalos *semiabiertos:*

$$(a, b] = \{x \in \mathbb{R} \mid a < x \leq b\}$$

y

$$[a, b) = \{ x \in \mathbb{R} \mid a \leq x < b\}$$

Lo que es más, se admiten $a = -\infty$ y $b = +\infty$ con el significado evidente siempre y cuando queden en el lado abierto de un intervalo.

Un *intervalo entero* es un conjunto de enteros que yacen entre dos límites. Sean i y j dos enteros tales que $i \leq j + 1$. El intervalo $[i..j]$ se indica por

$$\{n \in \mathbb{Z} \mid i \leq n \leq j\}$$

Este intervalo está vacío si $i = j + 1$. Obsérvese que $| [i .. j] | = j - i + 1$.

1.4.4 Funciones y relaciones

Sean X e Y dos conjuntos. Todo subconjunto ρ de su producto cartesiano $X \times Y$ es una *relación*. Cuando $x \in X$ e $y \in Y$, decimos que x está relacionado con y según ρ, lo cual se denota $x \rho y$, si y sólo si $(x, y) \in \rho$. Por ejemplo, uno puede pensar en la relación «≤» con respecto a los números enteros como el conjunto de los pares de enteros tales que el primer componente del par es menor o igual que el segundo.

Considérese cualquier relación f entre X e Y. La relación se llama *función* si, para cada $x \in X$, existe sólo un y perteneciente a Y tal que el par $(x,y) \in f$. Esto se representa con la expresión f: $X \rightarrow Y$, lo que se lee «f es una función de X en Y». Dado $x \in X$, el único $y \in Y$ tal que $(x, y) \in f$ se representa como $f(x)$. El conjunto X se llama *dominio* de la función, Y es su *imagen*, y el conjunto $f[X] = \{f(x) \mid x \in X\}$ es su *rango*. En general, $f[Z]$ denota $\{f(x) \mid x \in Z\}$ siempre que $Z \subseteq X$.

Una función f: $X \rightarrow Y$ se dice *inyectiva* (o bien uno a uno) si no existen dos $x_1, x_2 \in X$ tales que $f(x_1) = f(x_2)$. Es *suprayectiva* (o sobreyectiva) si para todos los $y \in Y$ existe al menos una $x \in X$ tal que $f(x) = y$. En otras palabras, es *suprayectiva* si su rango coincide con su imagen. Se dice que es *biyectiva* si es a la vez *inyectiva* y *suprayectiva*. Si f es *biyectiva*, denotaremos por f^{-1}, que se pronuncia «la inversa de f» a la función de Y en X que está definida por $f(f^{-1}(y)) = y$ para todos los $y \in Y$.

Dado cualquier conjunto X, una función P: $X \rightarrow \{verdadero, falso\}$ se llama un *predicado* sobre X. Existe una equivalencia natural entre predicados de X y subconjuntos de X: el subconjunto correspondiente a P es $\{x \in X \mid P(x)\}$. Cuando P es un predicado sobre X, diremos en algunas ocasiones que P es una propiedad de X. Por ejemplo, la imparidad es una propiedad de los enteros, que es *verdadera* para los enteros impares y *falsa* para los enteros pares. También existe una interpretación natural de las fórmulas booleanas en términos de predicado. Por ejemplo, uno puede definir un predicado P: $\{verdadero, falso\}^3 \rightarrow \{verdadero, falso\}$ mediante

$$P(p, q, r) = (p \wedge q) \vee (\neg q \wedge r)$$

en cuyo caso P (*verdadero, falso, verdadero*) = *verdadero*.

1.4.5 Cuantificadores

Los símbolos $\forall$ y $\exists$ se leen «para todo» y «existe», respectivamente. Para ilustrar esto, considérese un conjunto arbitrario X y una propiedad P sobre X. Escribimos $(\forall x \in X)[P(x)]$ para indicar que «todos los x de X tienen la misma propiedad P». De manera similar:

$$(\exists x \in X)[P(x)]$$

significa que «existe al menos un elemento de x en X que tiene la propiedad P». Finalmente, escribiremos $(\exists! x \in X)[P(x)]$ para significar «existe exactamente un x en X que tiene la propiedad P». Si X es el conjunto vacío, $(\forall x \in X)[P(x)]$ siempre es vacíamente verdadero, intente encontrar un contraejemplo si no está de

acuerdo, mientras que $(\exists\, x \in X)\ [P\ (x)]$ siempre es trivialmente falso. Considérense los tres ejemplos concretos siguientes:

$$(\forall n \in \mathbf{N})\left[\sum_{i=1}^{n} i = \frac{n(n+1)}{2}\right]$$

$$\left(\exists!\ \ n \in \mathbf{N}^{+}\right)\left[\sum_{i=1}^{n} i = n^2\right]$$

$$(\exists\, m, n \in \mathbb{N})\ \ [m > 1, n > 1 \text{ y } mn = 12573]$$

Estos ejemplos afirman que la bien conocida fórmula para la suma de los n primeros enteros es siempre válida (véase la Sección 1.7.2), que esta suma es también igual a un n^2 sólo para un valor entero positivo de n, y que 12.573 es un entero compuesto, respectivamente.

Se puede utilizar una *alternancia* de los cuantificadores en una sola expresión. Por ejemplo:

$$(\forall\, n \in \mathbb{N})\ (\exists\, m \in \mathbb{N})\ [m > n\ ]$$

dice que para todo número natural existe otro número natural mayor todavía. Cuando se utiliza la alternancia de cuantificadores, el orden en el cual se presentan los cuantificadores es importante. Por ejemplo, la afirmación $(\exists\, m \in \mathbb{N})(\forall\, n \in \mathbb{N})\ [m > n\ ]$ es evidentemente falsa: significaría que existe un entero m que es mayor que todos los números naturales (incluyendo el propio m).

Siempre y cuando el conjunto X sea infinito, resulta útil decir que no solamente existe un $x \in X$ tal que la propiedad de $P\ (x)$ es cierta, sino que además existen infinitos de ellos. El cuantificador apropiado en este caso es $\tilde{\exists}$. Por ejemplo, $(\tilde{\exists}\ n \in \mathbb{N})$ [n es primo]. Obsérvese que $\tilde{\exists}$ es más fuerte que $\exists$ pero más débil que $\forall$. Otro cuantificador útil, más fuerte que $\tilde{\exists}$ pero todavía más débil que $\forall$, es $\tilde{\forall}$, que se usa cuando una propiedad es válida en todos los casos salvo posiblemente para un número finito de excepciones.

Por ejemplo, $(\tilde{\forall}\, n \in \mathbb{N})$ [si n es primo, entonces n es impar] significa que los números primos siempre son impares, salvo posiblemente por un número finito de excepciones, en este caso hay solamente una excepción: 2 es a la vez primo y par.

Cuando estamos interesados en las propiedades de los números naturales, existe una definición equivalente para estos cuantificadores, y suele ser mejor pensar en ellos en consecuencia. Una propiedad P de los números naturales es cierta con infinita frecuencia, si, independientemente de lo grande que sea m, existe un $n \geq m$ tal que $P\ (n)$ es válido. De manera similar, la propiedad P es válida para todos los números naturales salvo posiblemente por un número finito de excepciones si existe un natural m tal que $P\ (n)$ es válido para todos los nú-

meros naturales $n \geq m$. En este último caso, diremos que «la propiedad P es cierta para todos los enteros *suficientemente grandes*». Formalmente:

$$(\overset{\infty}{\exists}\ n \in \mathbb{N})\ [P\ (n)] \text{ es equivalente a } (\forall\ m \in \mathbb{N})\ (\exists\ n \geq m)\ [P\ (n)]$$

mientras que

$$(\overset{\infty}{\forall}\ n \in \mathbb{N})\ [P\ (n)] \text{ es equivalente a } (\exists\ m \in \mathbb{N})\ (\forall\ n \geq m)\ [P\ (n)]$$

El *principio de dualidad* para los cuantificadores dice que «no es cierto que la propiedad P sea válida para todo $x \in X$ si y sólo si existe al menos un $x \in X$ para el cual la propiedad P no es válida». En otras palabras:

$$\neg(\forall\ x \in X)\ [P\ (x)] \text{ es equivalente a } (\exists\ x \in X)\ [\neg\ P\ (x)]$$

De manera similar:

$$\neg(\exists\ x \in X)\ [P\ (x)] \text{ es equivalente a } (\forall\ x \in X)\ [\neg\ P\ (x)]$$

El principio de dualidad también es válido para $\overset{\infty}{\forall}$ y $\overset{\infty}{\exists}$.

1.4.6 Sumas y productos

Considérese una función $f: \mathbb{N} \to \mathbb{R}$ y un entero $n \geq 0$. (Esto incluye $f: \mathbb{N} \to \mathbb{N}$ como caso especial.) La suma de los valores tomados por f sobre los n primeros números positivos se denota mediante

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots + f(n)$$

que se lee «la suma de los $f\ (i)$ cuando i va desde 1 hasta n »

En el caso en que $n = 0$, la suma se define como 0. Esto se generaliza en la forma evidente para denotar una suma cuando i va desde m hasta n siempre y cuando $m \leq n + 1$. En algunas ocasiones resulta útil considerar sumas condicionales. Si P es una propiedad de los enteros,

$$\sum_{P(i)} f(i)$$

denota la suma de $f\ (i)$ para todos los enteros i tal que sea válido $P\ (i)$. Esta suma puede no estar bien definida si involucra a un número infinito de enteros, podemos incluso utilizar una notación mixta tal como

$$\sum_{\substack{i=1 \\ P(i)}}^{n} f(i),$$

que denota la suma de los valores tomados por f para aquellos enteros que se encuentran entre 1 y n para los cuales es válida la propiedad P. Si no hay tales enteros, la suma es 0. Por ejemplo,

$$\sum_{\substack{i=1 \\ i \text{ impar}}}^{10} i = 1+3+5+7+9 = 25.$$

En la Sección 1.7.2. se puede encontrar más información acerca de las sumas.

El producto de los valores tomados por f sobre los n primeros enteros positivos se denota mediante

$$\prod_{i=1}^{n} f(i) = f(1) \times f(2) \times f(3) \times \ldots \times f(n),$$

lo cual se lee «el producto de los f (i) cuando i va desde 1 hasta n». En el caso $n = 0$, se define el producto como 1. Esta notación se generaliza en la misma forma que en la notación del sumatorio.

1.4.7 Miscelánea

Si $b \neq 1$ y x son números reales estrictamente positivos, entonces $\log_b x$, que se lee «el logaritmo en base b de x», se define como el número real y tal que $b^y = x$. Por ejemplo, $\log_{10} 1.000 = 3$. Obsérvese que aun cuando b y x deben ser positivos, no existe tal restricción para y. Por ejemplo, $\log_{10} 0{,}001 = -3$. Cuando la base b no está especificada, interpretamos que se trata de e = 2,7182818..., la base de los llamados logaritmos *naturales* (algunos autores toman la base 10 cuando no se especifica y denotan el logaritmo natural como «ln».) En Algoritmia, la base que se utiliza más a menudo para los logaritmos es 2, y merece una notación propia: «lg x» es la abreviatura de «$\log_2 x$.» Aun cuando supondremos que el lector está familiarizado con los logaritmos, recordemos las identidades logarítmicas más importantes:

$$\log_a (xy) = \log_a x + \log_a y,$$

$$\log_a x^y = y \log_a x,$$

$$\log_a x = \frac{\log_b x}{\log_b a},$$

y por último $x^{\log_b y} = y^{\log_b x}$

Recuérdese también que el «log log n» es el logaritmo del logaritmo de n, pero «$\log^2 n$» es el cuadrado del logaritmo de n.

Si x es un número real, $\lfloor x \rfloor$ representa el mayor entero que no es mayor que x, y se denomina el suelo de x. Por ejemplo, $\lfloor 3\ 1/2 \rfloor = 3$. Cuando x es positivo, $\lfloor x \rfloor$ es el entero que se obtiene descartando la parte fraccionaria de x si es que exis-

te. Sin embargo, cuando x es negativo y no es un entero en sí x es más pequeño que este valor por una unidad. Por ejemplo, $\lfloor -3^1/_2 \rfloor = -4$. De manera similar, definimos el *techo* de x, que se denota como $\lceil x \rceil$, como el menor entero que no es menor que x. Obsérvese que $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$ para todo x.

Si $m \geq 0$ y $n > 0$ son enteros, m/n denota como siempre el resultado de dividir m por n, lo cual no es necesariamente un entero. Por ejemplo, $7/2 = 3^1/_2$. Denotamos el cociente entero mediante el símbolo « $\div$ », por tanto $7 \div 2 = 3$. Formalmente, $m \div n = \lfloor m/n \rfloor$. También utilizamos **mod** para denotar el operador «módulo» que se define como

$$m \bmod n = m - n \times (m \div n)$$

En otras palabras, m mod n es el resto cuando m es dividido por n.

Si m es un entero positivo, denotamos el producto de los m primeros enteros positivos como $m!$, lo cual se lee *factorial de m*. Es natural definir $0! = 1$. Ahora bien $n! = n \times (n - 1)!$ para todos los enteros positivos n. Una aproximación útil del factorial es la que da la *fórmula de Stirling:* $n! \approx \sqrt{2\pi n}\,(n/e)^n$, en donde e es la base de los logaritmos naturales.

Si n y r son enteros tales que $0 \leq r \leq n$, se representa mediante $\binom{n}{r}$ el número de formas de seleccionar r elementos de un conjunto de cardinalidad n, sin tener en cuenta el orden en el cual hagamos nuestras selecciones; véase la Sección 1.7.3.

1.5 TÉCNICA DE DEMOSTRACIÓN 1: CONTRADICCIÓN

Ya hemos visto que puede existir toda una selección de algoritmos disponibles cuando nos preparamos para resolver un problema. Para decidir cuál es el más adecuado para nuestra aplicación concreta, resulta crucial establecer las propiedades matemáticas de los diferentes algoritmos, tal como puede ser el tiempo de ejecución como función del tamaño del ejemplar que haya que resolver. Esto puede implicar demostrar estas propiedades mediante una demostración matemática. Esta sección y la siguiente revisan dos técnicas de demostración que suelen ser útiles en Algoritmia: la demostración por contradicción y la demostración por inducción matemática.

La *demostración por contradicción*, que también se conoce como *prueba indirecta*, consiste en demostrar la veracidad de una sentencia demostrando que su negación da lugar a una contradicción. En otras palabras, supongamos que se desea demostrar la sentencia S. Por ejemplo, S podría ser «existe un número infinito de números primos». Para dar una demostración indirecta de S, se empieza por suponer que S es falso (o, equivalentemente, suponiendo que «no S» es verdadero). ¿Qué se puede deducir si, a partir de esa suposición, el razonamiento matemático establece la veracidad de una afirmación que es evidentemente falsa? Naturalmente, podría ser que el razonamiento en cuestión estuviera equivocado. Sin embargo, si el razonamiento es correcto, la única explica-

ción posible es que la suposición original es falsa. De hecho, solamente es posible demostrar la veracidad de una falsedad a partir de una hipótesis falsa.

Ilustraremos este principio con dos ejemplos, el segundo de los cuales es una sorprendente ilustración de que las pruebas indirectas nos dejan algunas veces un sabor algo amargo. Nuestro primer ejemplo es el que ya se ha mencionado, y que era conocido por los antiguos griegos (Proposición Vigésima del Libro IX de los *Elementos* de Euclides).

Teorema 1.5.1 (Euclides) *Existen infinitos números primos*

Demostración. Sea P el conjunto de los números primos. Supongamos para buscar una contradicción que P es un conjunto finito. El conjunto P no es vacío, porque contiene al menos el entero 2. Dado que P es finito y no está vacío, tiene sentido multiplicar todos sus elementos. Sea x ese producto, y sea y el valor $x + 1$. Consideremos el menor entero d que es mayor que 1 y que es el divisor de y. Este entero existe ciertamente por cuanto y es mayor que 1 y no exigimos que d sea distinto de y. Obsérvese en primer lugar que d en sí es primo, porque en caso contrario todo divisor propio de d dividiría también a y, y sería más pequeño que d, que estaría en contradicción con la definición de d. (¿Ha observado el lector que la sentencia anterior es, en sí misma, una demostración por contradicción, anidada en el esquema general?) Por tanto, de acuerdo con nuestra suposición de que P contiene absolutamente todos los números primos, d pertenece a P. Esto demuestra que d es también divisor de x, puesto que x es el producto de una colección de enteros que contiene a d. Hemos llegado a la conclusión de que d divide exactamente tanto a x como a y. Pero se recordará que $y = x + 1$. Por tanto, hemos obtenido un entero d más grande que 1 y que divide a dos enteros consecutivos x e y. Esto es claramente imposible: si realmente d divide a x, entonces la división de y por d dejará necesariamente 1 como resto. La conclusión ineludible es que la suposición original era igualmente imposible. Pero la suposición original era que el conjunto P de todos los primos es finito, y por tanto su imposibilidad indica que el conjunto P es, de hecho, infinito.

Para el lector que tenga una mentalidad constructiva (lo cual debería ser cierto para todos y cada uno de los estudiosos de Algoritmia), esta demostración del Teorema de Euclides se puede transformar en un *algoritmo*, aunque no sea muy eficiente, capaz de hallar un nuevo número primo dado cualquier conjunto finito de primos.

```
función Nuevoprimo (P : conjunto de enteros)
{El argumento P debería ser un conjunto de primos no vacío finito}
   x ← producto de los elementos de P
   y ← x + 1
   d ← 1
   repetir d ← d+1 hasta que d divide a y
   devolver d
```

La demostración de Euclides establece que el valor proporcionado por *Nuevoprimo (P)* es un número primo que no pertenece a *P*. ¿Pero quién necesita a Euclides cuando está escribiendo un algoritmo para esta tarea? ¿Por qué no utilizar el siguiente algoritmo, mucho más sencillo?

función *EvitarEuclides* (*P* : conjunto de enteros)
{El argumento *P* ha de ser un conjunto de primos no vacío finito}
$x \leftarrow$ el mayor elemento de *P*
repetir $x \leftarrow x + 1$ **hasta** que *x* es primo
devolver *x*

Es evidente que este segundo algoritmo proporciona como resultado un número primo que no pertenece a *P*, ¿verdad? La respuesta es sí, *siempre y cuando el algoritmo termine*. El problema es que *EvitarEuclides* quedaría en un bucle infinito si *P* contuviera el mayor de los números primos.

Naturalmente, esta situación no se puede dar porque no existe tal cosa como «el mayor número primo», pero se necesita la demostración de Euclides para establecer esto. En resumen, *EvitarEuclides* funciona, pero la demostración de su terminación no es inmediata. Por contraposición, el hecho de que *Nuevoprimo* siempre termine es evidente (en el caso peor terminará cuando *d* alcance el valor de *y*), pero el hecho de que proporcione un nuevo primo requiere demostración.

Acabamos de ver que a veces es posible transformar una demostración matemática en un algoritmo. Desafortunadamente, esto no siempre sucede cuando la demostración es por contradicción. Ilustraremos esto con un ejemplo elegante.

Teorema 1.5.2 *Existen dos números irracionales* x *e* y *tales que* x^y *es racional*

Demostración. Para realizar la demostración por contradicción supongamos que x^y es necesariamente irracional siempre que tanto *x* como *y* son irracionales. Es bien conocido que $\sqrt{2}$ es irracional (esto se sabía ya en tiempos de Pitágoras, que vivió incluso antes de Euclides). Hagamos que *z* tome el valor de $\sqrt{2}^{\sqrt{2}}$. Por nuestra suposición *z* es irracional por cuanto es el resultado de elevar un número irracional ($\sqrt{2}$) a una potencia irracional (una vez más $\sqrt{2}$). Hagamos ahora que *w* tenga el valor de $z^{\sqrt{2}}$. Una vez más, tenemos que *w* es irracional por nuestra suposición, en cuanto que tanto *z* como $\sqrt{2}$ son irracionales. Pero

$$w = z^{\sqrt{2}} = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = (\sqrt{2})^{(\sqrt{2}x\sqrt{2})} = (\sqrt{2})^2 = 2.$$

Hemos llegado a la conclusión de que 2 es irracional, lo cual claramente es falso. Por tanto hay que concluir que nuestra suposición era falsa: debe ser posible obtener un número racional cuando se eleva un número irracional a una potencia irracional.

Ahora bien, ¿cómo se podría transformar esta demostración en un algoritmo? Claramente el propósito del algoritmo sería mostrar dos números irracionales x e y tales que x^y es racional. A primera vista, puede uno sentirse tentado a decir que el algoritmo podría limitarse a indicar $x = z$ (tal como se define z en la demostración) e $y = \sqrt{2}$, puesto que se ha demostrado que z es irracional y que $z^{\sqrt{2}} = 2$. ¡Cuidado! la «demostración» de que z es irracional, depende de la suposición falsa con la que hemos comenzado y por tanto ésta demostración no es válida (sólo la demostración no es válida; a decir verdad, es cierto que z es irracional, pero esto es difícil de establecer). Siempre hay que tener cuidado de no utilizar posteriormente un resultado intermedio «demostrado» en medio de una demostración por contradicción.

No hay una forma directa de extraer la pareja requerida (x, y) a partir de la demostración del teorema. Lo que se puede hacer es extraer *dos* parejas y afirmar con confianza que una de ellas hace lo que deseamos (pero no será posible saber cuál). Esta demostración se denomina *no constructiva* y no es muy frecuente entre las pruebas indirectas. Aun cuando algunos matemáticos no aceptan las demostraciones no constructivas, la mayoría de ellos las ven como perfectamente válidas. En todo caso, nos abstendremos en lo posible de utilizarlas en el contexto de la Algoritmia.

1.6 TÉCNICA DE DEMOSTRACIÓN 2: INDUCCIÓN MATEMÁTICA

De las herramientas matemáticas básicas útiles en la Algoritmia, quizá no haya ninguna más importante que la *inducción matemática*. No sólo nos permite demostrar propiedades interesantes acerca de la corrección y eficiencia de los algoritmos, sino que además veremos en la Sección 1.6.4 que puede incluso utilizarse para *determinar* qué propiedades es preciso probar.

Antes de discutir la técnica, se ha de indicar una digresión acerca de la naturaleza del descubrimiento científico. En la ciencia hay dos enfoques opuestos fundamentales: *inducción* y *deducción*. De acuerdo con el *Concise Oxford Dictionary*, la inducción consiste en «inferir una ley general a partir de casos particulares», mientras que una deducción es una «inferencia de lo general a lo particular». Veremos que, aun cuando la inducción puede dar lugar a conclusiones falsas, no se puede despreciar. La deducción, por otra parte, siempre es válida con tal de que sea aplicada correctamente.

En general no se puede confiar en el resultado del razonamiento inductivo. Mientras que haya casos que no hayan sido considerados, sigue siendo posible que la regla general inducida sea incorrecta. Por ejemplo, la experiencia de todos los días nos puede haber convencido inductivamente que «siempre es posible meter una persona más en un tranvía». Pero unos instantes de pensamiento nos muestran que esta regla es absurda. Como ejemplo más matemático considérese el polinomio $p(n) = n^2 + n + 41$. Si computamos $p(0)$, $p(1)$, $p(2)$,..., $p(10)$, se va encontrando 41, 43, 47, 53, 61, 71, 83, 97, 113, 131 y 151. Es fácil verificar que todos estos enteros son números primos. Por tanto es natural inferir por *in-*

ducción que $p\ (n)$ es primo para todos los valores enteros de n. Pero de hecho $p\ (40) = 1.681 = 41^2$ es compuesto. Para un ejemplo geométrico bellísimo de falsa inducción, le recomendamos hacer el problema 1.19.

Un ejemplo más sorprendente de inducción incorrecta es el dado por una conjetura de Euler, que formuló en 1769. ¿Es posible que la suma de tres cuartas potencias sea una cuarta potencia? Formalmente, es posible encontrar cuatro enteros positivos A, B, C y D tales que

$$¿A^4 + B^4 + C^4 = D^4?$$

Al no poder encontrar ni siquiera un ejemplo de este comportamiento, Euler conjeturó que esta ecuación nunca se podría satisfacer. (Esta conjetura está relacionada con el Último Teorema de Fermat.) Transcurrieron más de dos siglos antes de que Elkies en 1987 descubriese el primer contraejemplo, que implicaba números de siete y ocho cifras. Posteriormente ha sido demostrado por parte de Frye, utilizando cientos de horas de tiempo de computación en varias Connection Machines, que el único contraejemplo en el que D es menor que un millón es

$$95.800^4 + 217.519^4 + 414.560^4 = 422.481^4$$

(sin contar con la solución obtenida multiplicando cada uno de estos números por 2). Obsérvese que 422.481^4 es un número de 23 dígitos.

La ecuación de Pell proporciona un caso todavía más extremo de razonamiento inductivo tentador pero incorrecto. Considérese el polinomio $p\ (n) = 991n^2 + 1$. La pregunta es si existe un entero positivo n tal que $p\ (n)$ es un cuadrado perfecto. Si uno va probando varios valores para n, resulta cada vez más tentador suponer inductivamente que la respuesta es negativa. Pero de hecho se puede obtener un cuadrado perfecto con este polinomio: la solución más *pequeña* se obtiene cuando

$$n = 12\,055\,735\,790\,331\,359\,447\,442\,538\,767$$

Por contraste, el razonamiento deductivo no está sometido a errores de este tipo. Siempre y cuando la regla invocada sea correcta, y sea aplicable a la situación que se estudie, la conclusión que se alcanza es necesariamente correcta. Matemáticamente, si es cierto que alguna afirmación $P(x)$ es válida para todos los x de algún conjunto X, y si de hecho y pertenece a X, entonces se puede afirmar sin duda que $P(y)$ es válido. No quiere decir esto que no se pueda inferir algo falso utilizando un razonamiento deductivo. A partir de una premisa falsa, se puede derivar deductivamente una conclusión falsa; éste es el principio que subyace a las demostraciones indirectas. Por ejemplo, si es correcto que $P(x)$ es cierto para todos los x de X, pero si somos descuidados al aplicar esta regla a algún y que no pertenezca a X, entonces podemos creer equivocadamente que $P(y)$ es cierto. De manera similar, si nuestra creencia en que $P(x)$ es cierto para todos los x de X está basada en

un razonamiento inductivo descuidado, entonces $P(y)$ puede resultar falso aun cuando de hecho y pertenezca a X. En conclusión, el razonamiento deductivo puede producir un resultado incorrecto, pero *sólo* si las reglas que se siguen son incorrectas o si no se siguen correctamente.

Como ejemplo perteneciente a la ciencia de la computación consideremos una vez más la multiplicación *à la russe*, descrita en la Sección 1.2. Si se prueba este algoritmo con varios pares de números positivos se observará que siempre da la respuesta correcta. Por inducción, puede uno formular la conjetura consistente en que el algoritmo siempre es correcto. En este caso, la conjetura alcanzada inductivamente resulta ser cierta: mostraremos rigurosamente (mediante razonamiento deductivo) la corrección de este algoritmo mediante el teorema 1.6.4. Una vez que se ha establecido la corrección, si se utiliza el algoritmo para multiplicar 981 por 1.234 y se obtiene 1.210.554, se puede concluir que

$$981 \times 1.234 = 1.210.554$$

En esta ocasión, la corrección de este caso específico de multiplicación de enteros es un caso especial de la corrección del algoritmo en general. Por tanto, la conclusión de que $981 \times 1.234 = 1.210.554$ está basada en un razonamiento deductivo. Sin embargo, la prueba de la corrección del algoritmo no dice nada acerca de su comportamiento para números negativos y fraccionarios y por tanto no se puede deducir nada acerca del resultado obtenido por el algoritmo si se ejecuta con –12 y 83,7

Es muy probable que el lector se pregunte en este momento por qué podría nadie utilizar la inducción, que es proclive a errores, en lugar de una deducción que es a prueba de bomba. Hay dos razones básicas para el uso de la inducción en el proceso del descubrimiento científico. Si uno es un físico cuyo objetivo es determinar las leyes fundamentales que gobiernan el Universo, es *preciso* utilizar un enfoque inductivo: las reglas que se infieren deberían de reflejar datos reales obtenidos de experimentos. Aun cuando sea uno un físico teórico, tal como Einstein, siguen siendo necesarios experimentos reales que otras personas hayan efectuado. Por ejemplo, Halley predijo el retorno de su cometa homónimo por razonamiento inductivo, y también por razonamiento inductivo Mendeleev predijo no sólo la existencia de elementos químicos todavía no descubiertos, sino también sus propiedades químicas.

Con seguridad, ¿sólo será legítima en matemáticas y en las rigurosas ciencias de la computación la deducción? Después de todo las propiedades matemáticas como el hecho consistente en que existen infinitos números primos (teorema 1.5.1) y que la multiplicación *à la russe* es un algoritmo correcto (teorema 1.6.4) se pueden demostrar de una forma deductiva rigurosa, sin datos experimentales. Los razonamientos inductivos deberían eliminarse de las matemáticas, ¿verdad? ¡Pues no! En realidad la matemática puede ser también una ciencia experimental. No es infrecuente que un matemático descubra una verdad matemática considerando varios casos especiales e infiriendo a partir de ellos por *inducción* una regla general que parece plausible. Por ejemplo, si se observa que

$$
\begin{aligned}
1^3 &= 1 = 1^2 \\
1^3 + 2^3 &= 9 = 3^2 \\
1^3 + 2^3 + 3^3 &= 36 = 6^2 \\
1^3 + 2^3 + 3^3 + 4^3 &= 100 = 10^2 \\
1^3 + 2^3 + 3^3 + 4^3 + 5^3 &= 225 = 15^2
\end{aligned}
$$

Se puede empezar a sospechar que la suma de los cubos de los números enteros positivos es siempre un cuadrado perfecto. Resulta en este caso que el razonamiento inductivo proporciona una ley correcta. Si soy todavía más perceptivo quizá me dé cuenta de que esta suma de los cubos es precisamente el cuadrado de la suma de los primeros números enteros positivos; véase el problema 1.2.1.

Sin embargo, independientemente de lo tentadora que sea la evidencia cuantos más valores de n se ensayan, una regla de este tipo no se puede basar en la evidencia inductiva solamente. La diferencia entre las matemáticas y las ciencias inherentemente experimentales es que, una vez se ha descubierto por inducción una ley matemática general, debemos demostrarla rigurosamente aplicando el proceso deductivo. Sin embargo, la inducción tiene su lugar en el proceso matemático. En caso contrario, ¿como podríamos esperar comprobar rigurosamente un teorema cuyo enunciado ni siquiera ha sido formulado? Para resumir, la inducción es necesaria para formular conjeturas, y la deducción es igualmente necesaria para demostrarlas, o a veces para demostrar su falsedad. Ninguna de estas técnicas puede ocupar el lugar de la otra. La deducción sólo es suficiente para las matemáticas «muertas» o congeladas, tal como en los *Elementos* de Euclides (que quizá sean el mayor monumento de la historia a las matemáticas deductivas, aun cuando no cabe duda de que gran parte de su material fue descubierto por razonamiento inductivo). Pero se necesita la inducción para mantener vivas las matemáticas. Tal como Pólya dijo una vez «las matemáticas presentadas con rigor son una ciencia deductiva sistemática, pero las matemáticas que se están haciendo son una ciencia inductiva experimental».

Por último, la moraleja de esta digresión: una de las técnicas *deductivas* más útiles que están disponibles en matemáticas tiene la mala fortuna de llamarse *inducción matemática*. Esta terminología resulta confusa, pero tenemos que vivir con ella.

1.6.1 El principio de inducción matemática

Considérese el algoritmo siguiente:

función *cuadrado* (n)
  Si $n = 0$ **entonces** devolver 0
  Si no devolver $2n$ + *cuadrado* ($n - 1$) – 1

Si se prueba con unas cuantas entradas pequeñas, se observa que:

cuadrado (0) = 0, *cuadrado* (1) = 1, *cuadrado* (2) = 4, *cuadrado* (3) = 9, *cuadrado* (4) = 16

Por inducción, parece evidente que *cuadrado* (n) = n^2 para todos los $n \geq 0$, ¿pero cómo podría demostrarse esto rigurosamente? ¿Será verdad? Diremos que el

algoritmo *tiene éxito* sobre el entero n siempre que *cuadrado* $(n) = n^2$, y que *fracasa* en caso contrario.

Considérese cualquier entero $n \geq 1$ y supóngase por el momento que el algoritmo tiene éxito en $n-1$. Por definición del algoritmo *cuadrado* $(n) = 2n +$ *cuadrado* $(n-1) - 1$. Por nuestra suposición *cuadrado* $(n-1) = (n-1)^2$. Por tanto:

$$\textit{cuadrado}\ (n) = 2n + (n-1)^2 - 1 = 2n + (n^2 - 2n + 1) - 1 = n^2$$

¿Qué es lo que hemos conseguido? Hemos demostrado que el algoritmo debe tener éxito en n siempre que tenga éxito en $n-1$, siempre y cuando $n \geq 1$. Además está claro que tiene éxito en $n = 0$.

El *principio de inducción matemática*, que se describe más adelante, nos permite inferir a partir de lo anterior que el algoritmo tiene éxito en todos los $n \geq 0$. Hay dos formas de comprender por qué se sigue esta conclusión: de forma constructiva y por contradicción. Considérese cualquier entero positivo m sobre el cual se desea probar que el algoritmo tiene éxito. A efectos de nuestra discusión, supongamos que $m \geq 9$ (los valores menores se pueden demostrar fácilmente). Ya sabemos que el algoritmo tiene éxito en 4. A partir de la regla general consistente en que debe tener éxito en n siempre que tenga éxito en $n-1$ para $n \geq 1$, inferimos que también tendrá éxito en 5. Aplicando una vez más esta regla se muestra que el algoritmo tiene éxito en 6 también. Puesto que tiene éxito en 6, también tiene que tener éxito en 7, y así sucesivamente. Este razonamiento continúa tantas veces como sea necesario para llegar a la conclusión de que el algoritmo tiene éxito en $m-1$. Finalmente, dado que tiene éxito en $m-1$, tiene que tener éxito en m también. Está claro que podríamos efectuar este razonamiento de forma explícita, sin necesidad de «y así sucesivamente», para cualquier valor positivo fijo de m.

Si preferimos una única demostración que funcione para todos los $n \geq 0$ y que no contenga «y así sucesivamente», debemos aceptar el *axioma del mínimo entero*, que dice que todo conjunto no vacío de enteros positivos contiene un elemento mínimo; véase el problema 1.24. Este axioma nos permite utilizar este número mínimo como el fundamento a partir del cual demostraremos teoremas.

Ahora, para demostrar la corrección (exactitud) del algoritmo por contradicción, supongamos que existe al menos un entero positivo en el cual falla el algoritmo. Sea n el menor de estos enteros, que existe por el axioma del mínimo entero. En primer lugar, n tiene que ser mayor o igual a 5, puesto que ya hemos verificado que *cuadrado* $(i) = i^2$ cuando $i = 1, 2, 3$ ó 4. En segundo lugar el algoritmo tiene que tener éxito en $n-1$, porque en caso contrario n no sería el menor entero positivo en el cual falla. Pero esto implica por regla general que el algoritmo también tiene éxito en n, lo cual contradice a nuestra suposición acerca de la selección de n. Por tanto esa n no puede existir, lo cual significa que el algoritmo tiene éxito para todos los enteros positivos. Puesto que también sabemos que el algoritmo tiene éxito en 0, concluiremos que *cuadrado* $(n) = n^2$ para todos los enteros de $n \geq 0$.

Ahora vamos a indicar una versión sencilla del principio de inducción matemática, que es suficiente en muchos casos. En la Sección 1.6.3 se da una versión más potente de este principio. Considérese cualquier propiedad P de los enteros. Por ejemplo, $P(n)$ podría ser «*cuadrado* $(n) = n^2$ o «la suma de los cubos de

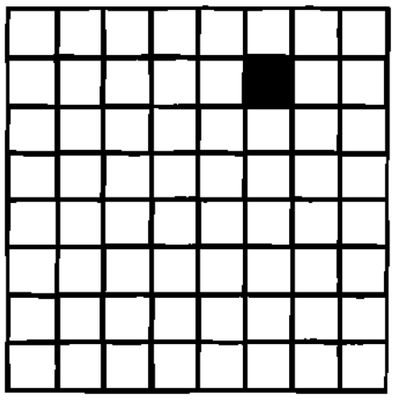
los n primeros números enteros es igual al cuadrado de la suma de esos enteros», o «$n^3 < 2^n$». Las dos primeras propiedades son ciertas para todos los $n \geq 0$, mientras que la tercera es válida siempre y cuando $n \geq 10$. Considérese también un entero a, que se conoce con el nombre de *base*. Si

1. $P(a)$ es cierto
2. $P(n)$ debe de ser cierto siempre que $P(n-1)$ sea válido para todos los enteros $n > a$

entonces la propiedad $P(n)$ es válida para todos los enteros $n \geq a$. Usando este principio, podríamos afirmar que *cuadrado* $(n) = n^2$ para todos los $n \geq 0$, e inmediatamente, demostrar que *cuadrado* $(0) = 0 = 0^2$ y que *cuadrado* $(n) = n^2$ siempre que *cuadrado* $(n-1)^2$ y $n \geq 1$.

Nuestro primer ejemplo de inducción matemática mostraba cómo se puede utilizar de forma rigurosa para demostrar la corrección de un algoritmo. Como segundo ejemplo, vamos a ver cómo las demostraciones mediante inducción matemática se pueden *transformar* a veces en algoritmos. Este ejemplo también es instructivo por cuanto pone de manifiesto la forma correcta de escribir una demostración por inducción matemática. La discusión que sigue hace hincapié en los puntos importantes que son comunes a este tipo de pruebas.

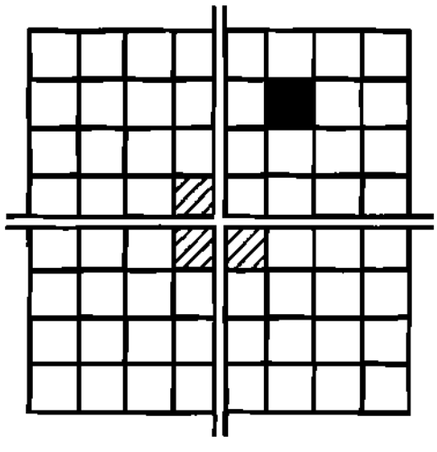
Considérese el siguiente problema de *embaldosado*. Se nos da un tablero dividido en cuadrados iguales. Hay m cuadrados por fila y m cuadrados por columna, en donde m es una potencia de 2. Un cuadrado arbitrario del tablero se distingue como *especial;* véase la Figura 1.5 (a).



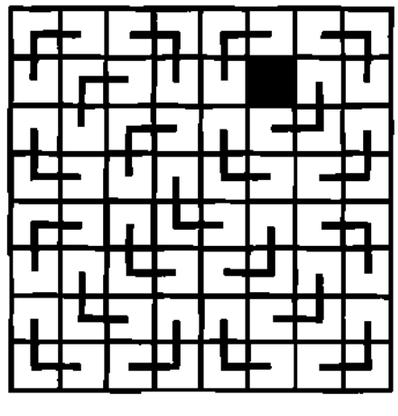
(a) Tablero con un cuadrado especial



(b) Una baldosa



(c) Colocada la primera baldosa



(d) Solución

Figura 1.5. El problema de embaldosado

También se nos da un montón de baldosas, cada una de las cuales tiene el aspecto de un tablero 2×2 del cual se ha eliminado un cuadrado, según se ilustra en la figura 1.5 (b). El acertijo consiste en recubrir el tablero con estas baldosas, para que cada cuadrado quede cubierto exactamente una vez con excepción del cuadrado especial, que quedará en blanco. Este recubrimiento se denomina *embaldosado*. La figura 1.5 (d) da una solución del caso dado en la figura 1.5 (a).

Teorema 1.6.1 *El problema de teselado siempre se puede resolver*

Demostración. La demostración se hace por inducción matemática sobre el entero n tal que $m = 2^n$.

◊ *Base*: el caso $n = 0$ se satisface de forma trivial. Aquí $m = 1$, y el «tablero» 1×1 es un único cuadrado, que es necesariamente especial. ¡Este tablero se recubre sin hacer nada! (Si no le gusta este argumento, compruebe el caso siguiente por orden de sencillez: si $n = 1$, entonces $m = 2$ y todo tablero 2×2 del cual se elimine un cuadrado tiene exactamente el aspecto de una baldosa, por definición.)

◊ *Paso de inducción*: considérese cualquier $n \geq 1$. Sea $m = 2^n$. Asúmase la *hipótesis de inducción*, consistente en que el teorema es válido para tableros $2^{n-1} \times 2^{n-1}$. Considérese un tablero $m \times m$, que contiene un cuadrado especial colocado arbitrariamente. Divídase el tablero por cuatro subtableros iguales partiéndolo en dos horizontal y verticalmente. El cuadrado especial original pertenece ahora exactamente a uno de los subtableros. Colóquese una baldosa en medio del tablero original de tal manera que cubra exactamente un cuadrado de cada uno de los demás subtableros; véase la figura 1.5 (*c*). Consideremos cada uno de los tres cuadrados así cubiertos «especial» para el subtablero correspondiente. Nos quedan cuatro subtableros del tipo $2^{n-1} \times 2^{n-1}$, cada uno de los cuales contiene un cuadrado especial. Por nuestra hipótesis de inducción, cada uno de estos cuatro subtableros se puede recubrir. La solución final se obtiene combinando los recubrimientos de los subtableros junto con la baldosa colocada en la posición media del tablero original.

Puesto que el teorema es verdadero cuando $m = 2^0$, y dado que su validez para $m = 2^n$ se sigue de la suposición de su validez para $m = 2^{n-1}$ para todos los $n \geq 1$, se sigue del principio de inducción matemática que el teorema es verdadero para todo m, siempre y cuando m sea una potencia de 2.

El lector no debería tener dificultad para transformar esta prueba de un teorema matemático a un algoritmo para efectuar el embaldosamiento real (quizá no sea un algoritmo de computadora, pero es al menos un algoritmo adecuado para el «procesamiento a mano»). Este algoritmo de teselado sigue el esquema general conocido con el nombre de divide y vencerás, que ya habíamos encontrado en la Sección 1.2 y que estudiaremos detenidamente en el Capítulo 7. Esta situación no es infrecuente cuando se demuestra constructivamente un teorema por inducción matemática.

Examinemos ahora con detalle todos los aspectos de una demostración formalmente correcta por inducción matemática, tal como la anterior. Considérese una vez más una propiedad abstracta P de los enteros, un entero a, y supóngase que se desea demostrar que $P(n)$ es válido para todos los $n \geq a$. Es preciso comenzar la demostración mediante el *caso base*, que consiste en demostrar que $P(a)$ es válido. Este caso base suele ser sencillo, y algunas veces incluso trivial, pero resulta crucial efectuarlo correctamente; en caso contrario, toda la «demostración» carece literalmente de fundamento.

El caso base va seguido por el *paso de inducción*, que suele ser más complicado. Éste debería empezar por «considérese cualquier $n > a$»(o de manera equivalente «consideremos cualquier $n \geq a + 1$»). Debería continuar con una indicación explícita de la hipótesis de inducción, que establece esencialmente que $P(n-1)$ es válido. En este momento queda por demostrar que es posible inferir que $P(n)$ es válido suponiendo válida la *hipótesis de inducción*. Por último, se puede insertar una frase adicional tal como la presente al final de la demostración del teorema 1.6.1, pero esto no suele ser necesario.

Con respecto a la hipótesis de inducción, es importante comprender que suponemos que $P(n-1)$ es válido de forma *provisional*; no sabemos realmente que sea válido mientras no se haya demostrado el teorema. En otras palabras, el objetivo del paso de inducción es demostrar que la veracidad de $P(n)$ se deduce lógicamente de la veracidad de $P(n-1)$, independientemente de si $P(n-1)$ es válido. De hecho si $P(n-1)$ no es válido, el paso de inducción no nos permite llegar a ninguna conclusión acerca de la veracidad de $P(n)$.

Por ejemplo, considérese la afirmación «$n^3 < 2^n$», que representaremos mediante $P(n)$. Para un positivo entero n, resulta sencillo demostrar que $n^3 < 2 \times (n-1)^3$ precisamente si $n \geq 5$. Considérese cualquier $n \geq 5$ y asumamos provisionalmente que $P(n-1)$ es verdadero. Ahora

$$\begin{aligned} n^3 &< 2 \times (n-1)^3 && \text{porque } n \geq 5 \\ &< 2 \times 2^{n-1} && \text{por la suposición consistente en que } P(n-1) \text{ es válido} \\ &= 2^n. \end{aligned}$$

De esta manera se ve que $P(n)$ se sigue lógicamente de $P(n-1)$ siempre que $n \geq 5$. Sin embargo, $P(4)$ no es válido (se diría que $4^3 < 2^4$, lo cual sería decir que $64 < 16$) y por tanto no se puede inferir nada con respecto a la veracidad de $P(5)$. Por prueba y error averiguaremos sin embargo que $P(10)$ sí es válido ($10^3 = 1000 < 2^{10} = 1024$). Por consiguiente es *legítimo* inferir que $P(11)$ también es válido, y de la veracidad de $P(11)$ se sigue que $P(12)$ también es válido, y así sucesivamente. Por el principio de inducción matemática, dado que $P(10)$ es válido y dado que $P(n)$ se sigue de $P(n-1)$ siempre que $n \geq 5$, concluimos que $n^3 < 2^n$ es verdadero para todo $n \geq 10$. Resulta instructivo observar que $P(n)$ también es válido para $n = 0$ y $n = 1$, pero no podemos utilizar estos puntos como base de la inducción matemática porque el paso de inducción no es aplicable para valores tan pequeños de n.

Quizá suceda que la propiedad que deseamos demostrar no afecta a todos los enteros mayores o iguales a uno dado. Nuestro acertijo del embaldosado concierne solamente al conjunto de enteros que son potencias de 2. En algunas ocasiones, la propiedad no concierne en absoluto a los números enteros. Por ejemplo, no es infrecuente en Algoritmia tener la necesidad de demostrar una propiedad de los grafos. (Se podría decir, incluso, que nuestro problema del embaldosado no concierne *realmente* a los números enteros, sino más bien a los tableros y a las baldosas, pero esto sería partir un pelo en el aire.) En tales casos, si se ha de utilizar la inducción matemática sencilla, la propiedad que hay que demostrar deberá transformarse primeramente en una propiedad del conjunto de todos los números enteros que no sean menores que un cierto caso base (un enfoque alternativo se ofrece en la Sección 1.6.3). En nuestro ejemplo de embaldosado, demostrábamos que $P(n)$ es válido para todas las potencias de 2, demostrando que $Q(n)$ es válido para $n \geq 0$, en donde $Q(n)$ es equivalente a $P(2^n)$. Cuando esta transformación es necesaria, se acostumbra comenzar la demostración (tal como hicimos) con las palabras «la demostración es por inducción matemática *sobre tal y cual parámetro*». De esta manera se realizan demostraciones acerca del número de nodos de un grafo, acerca de la longitud de una cadena de caracteres, acerca de la profundidad de un árbol, y así sucesivamente.

Existe un aspecto de las demostraciones por inducción matemática que la mayoría de los principiantes encuentra sorprendente, si es que no les parece paradójico: *¡a veces es más sencillo demostrar una afirmación más fuerte que una más débil*! Ilustraremos esto con un ejemplo que ya hemos encontrado. Vimos que resulta sencillo conjeturar por inducción (no por inducción *matemática*) que la suma de los cubos de los n primeros números enteros siempre es un cuadrado perfecto. Demostrar esto por inducción matemática no es sencillo. La dificultad estriba en que una hipótesis de inducción tal como «la suma de los cubos de los $n-1$ primeros números enteros es un cuadrado perfecto» no sirve de gran ayuda para demostrar que esto también sucede para los n primeros números enteros porque no dice *cuál* de los cuadrados es perfecto: en general, no hay ninguna razón para creer que se obtenga un cuadrado cuando n^3 se suma con otro cuadrado. Por contraste, resulta más sencillo demostrar el teorema más fuerte consistente en que nuestra suma de cubos es precisamente el cuadrado de la suma de los n primeros enteros: la hipótesis de inducción es ahora mucho más significativa; véase el problema 1.21.

1.6.2 Un asunto completamente distinto

El error más común en el diseño de demostraciones mediante inducción matemática se merece una subsección propia. Consideremos el siguiente teorema absurdo.

Teorema 1.6.2 *Todos los caballos son del mismo color*

Demostración. Demostraremos que todo conjunto de caballos contiene solamente caballos del mismo color. En particular esto será cierto para el conjunto de todos los caballos. Sea H un conjunto arbitrario de caballos. Demostremos por inducción matemática sobre el número n de caballos que hay en H que son todos ellos del mismo color.

◊ *Base*: el caso $n = 0$ es trivialmente cierto: si no hay caballos en H entonces son ciertamente todos del mismo color. (Si no le gusta el argumento, compruebe el siguiente caso sencillo: si $n = 1$, entonces hay un solo caballo en H, y una vez más es evidentemente cierto que son «todos» del mismo color.)

◊ Paso de inducción: considérese cualquier número n de caballos en H. Llámense estos caballos $h_1, h_2, \ldots, h_n$. Supongamos que la hipótesis de inducción consiste en que todo conjunto de $n - 1$ caballos contiene solamente caballos de un solo color (pero por supuesto los caballos de un conjunto podrían a priori ser diferentes de los caballos de otro conjunto). Sea H_1 el conjunto que se obtiene eliminando el caballo h_1 de H, y definamos H_2 de forma similar; véase la figura 1.6:

H_1:		h_2	h_3	h_4	h_5
H_2:	h_1		h_3	h_4	h_5

Figura 1.6. Caballos del mismo color (n = 5)

En cada uno de estos dos nuevos conjuntos hay $n - 1$ caballos. Por tanto, la hipótesis de inducción es aplicable a ellos. En particular, todos los caballos de H_1 son del mismo color, digamos c_1, y todos los caballos de H_2 son también de un único (posiblemente distinto) color, digamos c_2. Pero ¿es realmente posible que el color c_1 sea diferente del color c_2? Ciertamente no, el caballo h_n, pertenece a los dos conjuntos por tanto ambos, c_1 y c_2, deben ser del mismo color que el de dicho caballo! Como todos los caballos de H pertenecen a cualquier H_1 o H_2 (o a ambos), concluimos que todos son del mismo color $c = c_1 = c_2$. Esto completa el paso de inducción y la demostración por inducción matemática.

Antes de seguir adelante, encuentre la falacia de la «demostración» anterior. Si piensa que el problema es que nuestra hipótesis de inducción («todo conjunto de $n - 1$ caballos debe de contener solamente caballos de un mismo color») era absurda, ¡piénselo de nuevo!

Solución: El problema es que «h_n pertenece a ambos conjuntos» *no* es cierto para $n = 2$, ¡puesto que h_2 no pertenece a H_2! Nuestro razonamiento era impecable para los casos básicos $n = 0$ y $n = 1$. Además, es cierto que nuestro teorema es consecuencia válida para los conjuntos de n caballos, suponiendo que sea válido para $n - 1$, *pero solamente cuando* $n \geq 3$. Podemos pasar de 2 a 3 y de 3 a 4, y así sucesivamente, pero no de 1 a 2. Dado que los casos básicos son únicamente 0 y 1, y puesto que no podemos pasar de 1 a 2, el paso de inducción no puede comenzar. Este pequeño eslabón perdido dentro de la demostración basta para hacerla completamente inadmisible. Encontramos una situación similar al demostrar que $n^3 < 2^n$: el paso de inducción no era aplicable para $n < 5$, y por tanto la veracidad de la afirmación para $n = 0$ y $n = 1$ era irrelevante. La diferencia importante era que $n^3 < 2^n$ es cierto para $n = 10$, y por tanto también lo es para todos los valores mayores de n.

1.6.3 Inducción matemática generalizada

El principio de inducción matemática descrito hasta el momento es adecuado para demostrar muchas afirmaciones interesantes. Existen algunos casos, sin embargo, en los cuales es preferible un principio algo más potente. Se conoce con el nombre de inducción matemática *generalizada*. La situación se ilustra mediante el ejemplo siguiente:

Supongamos que se desea demostrar que todo entero compuesto se puede expresar como un producto de números primos. (El *teorema fundamental de la aritmética* nos dice que esta descomposición es única; esto *no* es lo que estamos intentando demostrar aquí.) No nos vamos a preocupar todavía por las bases de la inducción matemática, sino que vamos a saltar directamente al paso de inducción. Cuando se intenta demostrar que n se puede expresar como un producto de números primos (suponiendo que sea compuesto), la hipótesis de inducción «natural» sería que también se puede descomponer de esta manera $n - 1$. Sin embargo, desafiamos al lector a que encuentre algo en la descomposición de $n - 1$ en números primos que pueda ser útil o relevante para la descomposición de n en primos. Lo que se necesita realmente es una hipótesis de inducción más fuerte, consistente en que *todo* número compuesto entero menor que n se puede descomponer en un producto de números primos. La demostración correcta de nuestro teorema se da más adelante como teorema 1.6.3, después de expresar formalmente el principio de inducción matemática generalizada.

Otra generalización útil concierne a la base. A veces resulta necesario demostrar una base *extendida*, esto es, demostrar la base en más de un punto. Obsérvese que hemos demostrado bases extendidas para la corrección del algoritmo cuadrado y para el problema del embaldosado, pero era un lujo: el paso de inducción, en realidad, se podría haber aplicado al caso $n = 1$ a partir de la base $n = 0$. Esto no siempre va a suceder así: en algunas ocasiones, es preciso demostrar independientemente la validez de varios puntos de partida para que pueda despegar el paso de inducción. Encontraremos varios casos de este comportamiento más adelante en el libro; véanse por ejemplo los problemas 1.27 y 1.28.

Ya estamos preparados para formular un principio más general de inducción matemática. Considérese cualquier propiedad P de los enteros, y dos enteros a y b tales que $a \leq b$. Si:

1. $P(n)$ es válido para todo $a \leq n \leq b$, y
2. para cualquier entero $n \geq b$, el hecho consistente en que $P(n)$ es válido se sigue de la suposición consistente de que $P(m)$ es válido para todo m tal que $a \leq m < n$,

entonces la propiedad $P(n)$ es válida para todos los n tales que $n \geq a$.

Hay otra generalización adicional del principio de inducción matemática, que resulta cómodo cuando no estamos interesados en demostrar una afirmación para *todos* los enteros que no sean menores que la base. Suele suceder que necesitamos probar que es válida alguna propiedad P, pero sólo para aquellos enteros para los cuales es válida alguna otra propiedad Q. Ya hemos visto dos ejemplos de esta situación: el problema del embaldosado sólo es aplicable cuando m es una potencia de 2, y nuestra afirmación acerca de la descomposición en números pri-

mos sólo es aplicable a los números compuestos (aunque se podría extender de forma natural a los números primos). Cuando esto sucede, basta mencionar Q explícitamente en el enunciado del teorema que haya que demostrar, demostrar la base (posiblemente extendida) sólo para puntos en los que Q sea aplicable, y demostrar el paso de inducción solamente también en esos puntos. Por supuesto, la hipótesis de inducción se debilitará de forma similar. Considere cualquier n que sea mayor que la base tal que $Q(n)$ sea válido. Para demostrar que $P(n)$ es válido, tenemos perfecto derecho a suponer que $P(m)$ es válido cuando $a \leq m < n$, y cuando $Q(m)$ también es válido. En nuestro ejemplo del teselado, se nos permitía utilizar la hipótesis de inducción para recubrir tableros 4×4 cuando se está demostrando que se puede recubrir un tablero 8×8, pero *no* se nos permite suponer que se puede recubrir un tablero 5×5.

Antes de ilustrar este principio, obsérvese que permite que la base no exista. Esto sucede cuando $a = b$ porque en ese caso no hay enteros n tales que $a \leq n < b$. También puede ocurrir cuando $a < b$ si $Q(n)$ nunca es válido cuando $a \leq n < b$. Esto no invalida la demostración porque en tal caso la validez de $P(n)$ para el mínimo n al cual sea aplicable el paso de inducción se demuestra empleando una hipótesis de inducción inexistente, lo cual quiere decir que se demuestra sin hacer ninguna suposición. Nuestro primer ejemplo ilustra esto. El segundo muestra la forma de demostrar la corrección de un algoritmo por inducción matemática generalizada.

Teorema 1.6.3 *Todo entero positivo compuesto se puede expresar como un producto de números primos*

Demostración. La demostración se hace por inducción matemática generalizada. En este caso, no se necesita una base.

◊ *Paso de inducción*: considere cualquier entero compuesto $n \geq 4$. (Observe que 4 es el menor entero positivo compuesto, así que no tendría sentido considerar valores más pequeños de n.) Supongamos, como hipótesis de inducción, que todo entero compuesto positivo menor que n se puede expresar como un producto de números primos. (En el caso más pequeño, $n = 4$, esta hipótesis de inducción es irrelevante.) Considérese el menor entero d que sea mayor que 1 y que sea un divisor de n. Tal como se argumentaba en la demostración del teorema 1.5.1, d es necesariamente primo. Sea $m = n/d$. Observe que $1 < m < n$ porque n es compuesto y $d > 1$. Se tienen dos casos:

— Si m es primo, hemos descompuesto n como el producto de dos primos: $n = d \times m$.

— Si m es compuesto, es positivo y menor que n, y por tanto es aplicable la hipótesis de inducción: m se puede expresar como un producto de números primos, digamos $p_1 p_2 \cdots p_k$ Por tanto, $n = d \times m$ se puede expresar en la forma $n = dp_1 p_2 \cdots p_k$ que es un producto de números primos.

En todo caso, esto completa la demostración del paso de inducción y por tanto del teorema.

Hasta el momento, la hipótesis de inducción siempre se ocupaba de un conjunto finito de casos (para la inducción matemática sencilla, un solo caso; para la inducción matemática generalizada, muchos casos en general pero a veces ninguno). En nuestro ejemplo final de demostración por inducción matemática generalizada, la hipótesis de inducción abarca una infinidad de casos ¡a pesar de que se demuestre el paso de inducción en un caso finito! Esta vez, demostraremos que la multiplicación *à la russe* multiplica correctamente cualquier pareja de enteros positivos. La observación clave es la tabla que se produce cuando se multiplica 490 por 2.468, es casi idéntica a la figura 1.2, que se utilizaba para multiplicar 981 por 1.234. Las únicas diferencias son que falta la primera línea cuando se multiplica 490 por 2.468, y que por tanto el término 1.234 que se hallaba en aquella primera línea no se añade al resultado final; véase la figura 1.7. ¿Cuál es la relación entre los casos (981 1.234) y (490 2.468)? Desde luego, es que 490 = 981 ÷ 2 y que 2.468 = 2 × 1.234.

981	1.234	1.234			
490	2.468		490	2.468	
245	4.936	4.936	245	4.936	4.936
122	9.872		122	9.872	
61	19.744	19.744	61	19.744	19.744
30	39.488		30	39.488	
15	78.976	78.976	15	78.976	78.976
7	157.952	157.952	7	157.952	157.952
3	315.904	315.904	3	315.904	315.904
1	631.808	631.808	1	631.808	631.808
		1.210.554			1.209.320

Figura 1.7 Demostración de la multiplicación à la russe

> Teorema 1.6.4 *La multiplicación à la russe multiplica correctamente cualquier pareja de números enteros positivos*

Demostración. Supongamos que se desea multiplicar m por n. La demostración se hace por inducción matemática sobre el valor de m.

◊ *Base*: el caso $m = 1$ es fácil; sólo se tiene una fila que consta de un 1 en la columna de la izquierda y de n en la columna de la derecha. Esta fila no se tacha porque 1 no es par. Cuando se «suma» el único número que «queda» en la columna de la derecha, obtenemos evidentemente n, que es el resultado correcto de multiplicar 1 por n.

◊ *Paso de inducción*: considérese cualquier $m \geq 2$ y cualquier entero positivo n. Supongamos como hipótesis de inducción que la multiplicación *à la russe* multiplica correctamente s por t para cualquier entero positivo s menor que m y para cualquier entero positivo t. (Obsérvese que *no* se exige que t sea menor que n.) Hay que considerar dos casos:

— Si m es par, la segunda fila de la tabla obtenida cuando se multiplica m por n contiene $m/2$ en la columna de la izquierda, y $2n$ en la columna de la derecha. Esto es idéntico a la primera fila que se obtiene cuando se multiplica $m/2$ por $2n$. Dado que toda fila no inicial de estas tablas depende solamente de la fila anterior, la tabla obtenida cuando se multiplica m por n es por tanto idéntica a la que se obtiene al multiplicar $m/2$ por $2n$, salvo por la primera fila adicional, que contiene m en la columna de la izquierda y n en la columna de la derecha. Como m es par, esta fila adicional se tachará antes de la suma final. Por tanto, el resultado final obtenido cuando se multiplica m por n *à la russe* es el mismo que cuando se multiplica $m/2$ por $2n$. Pero $m/2$ es positivo y menor que m. Por tanto, la hipótesis de inducción es aplicable: el resultado obtenido cuando se multiplica $m/2$ por $2n$ *à la russe* es $(m/2) \times (2n)$, tal como debe de ser. Consiguientemente, el resultado que se obtiene cuando se multiplica m por n *à la russe* es igual a $(m/2) \times (2n) = mn$, que es lo que tiene que ser.
— El caso en que m es impar resulta similar, salvo que hay que sustituir $m/2$ por $(m-1)/2$ en todas partes, y que no se borra la primera fila cuando se multiplica m por n. Por tanto, el resultado final al multiplicar m por n *à la russe* es igual a n más el resultado de multiplicar $(m-1)/2$ por $2n$ *à la russe*. Por hipótesis de inducción, esto último se calcula correctamente en la forma $((m-1)/2) \times 2n$, y por tanto lo anterior se calcula en la forma $n + ((m-1)/2) \times 2n$, que es mn tal como debía de ser.

Esto completa la demostración del paso por inducción y por tanto del teorema.

1.6.4 Inducción constructiva

La inducción matemática se utiliza sobre todo como técnica de demostración. Con harta frecuencia, se utiliza para demostrar afirmaciones que parecen haber surgido de la nada, como quien saca un conejo de un sombrero. Aunque la veracidad de tales afirmaciones queda comprobada, su origen sigue siendo un misterio. Sin embargo, la inducción matemática es una herramienta tan potente que nos permite no sólo descubrir meramente la veracidad de un teorema, sino también su enunciado exacto. Al aplicar la técnica de *inducción constructiva* que se describe en esta sección, puede uno demostrar simultáneamente la veracidad de una afirmación hecha de forma imprecisa y también descubrir las especificaciones omitidas, gracias a las cuales es correcta esa afirmación. Ilustraremos esta técnica mediante dos ejemplos que hacen uso de la *sucesión de Fibonacci* que se define más adelante. El segundo ejemplo muestra la forma en que esta técnica puede resultar útil para el análisis de algoritmos.

La sucesión que lleva el nombre de Fibonacci, matemático italiano del Siglo XII, se presenta tradicionalmente en términos de conejos (aunque en este caso no salgan de un sombrero). Ésta es la forma en que el propio Fibonacci la presentara en su *Liberabaci*, publicado en 1202. Supongamos que una pareja de conejos da lugar a dos descendientes por mes. Los descendientes, a su vez, comienzan a reproducirse dos meses después, y así sucesivamente. Entonces, si compramos una pareja de conejos en el mes 1, seguiremos teniendo una pareja en el mes 2. En el mes 3 empezarán a reproducirse, así que ahora tendremos tres parejas; en

el mes 5, tanto ellos como su primera pareja de descendientes tendrán conejillos, así que tendremos cinco parejas, y así sucesivamente. Si no se muere ningún conejo, el número de parejas que se tendrá cada mes viene dado por los términos de la sucesión de Fibonacci, que se define de manera más formal mediante la recurrencia siguiente:

$$\begin{cases} f_0 = 0; f_1 = 1 & \text{y} \\ f_n = f_{n-1} + f_{n-2} & \text{para } n \geq 2 \end{cases}$$

La sucesión comienza en la forma 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... Tiene numerosas aplicaciones en ciencias de la computación, en matemáticas, y en la teoría de juegos. De Moivre obtuvo la fórmula siguiente, que es fácil demostrar por inducción matemática (véase el problema 1.27):

$$f_n = \frac{1}{\sqrt{5}}\left[\phi^n - (-\phi)^{-n}\right]$$

en donde $\phi = (1+\sqrt{5})/2$ es la *razón áurea*. Dado que es $0 < \phi^{-1} < 1$, el término $(-\phi)^{-n}$ se puede despreciar cuando n es grande. Por tanto, el valor de f_n es aproximadamente $\phi^n / \sqrt{5}$, que es exponencial en n.

Pero ¿de dónde viene la fórmula de De Moivre? En la Sección 4.7 veremos una técnica general para resolver recurrencias similares a la de Fibonacci. Mientras tanto, suponga que no sabe nada acerca de estas técnicas, y que tampoco conoce la fórmula De Moivre, pero que desea tener una idea del comportamiento de la sucesión de Fibonacci. Si calcula unos cuantos términos de la sucesión de Fibonacci, observará que crece con bastante rapidez (f_{100} es un número con 21 dígitos). Por tanto, la conjetura «la sucesión de Fibonacci crece exponencialmente» es razonable. ¿Cómo lo demostraría usted? La dificultad consiste en que esta conjetura es demasiado vaga para demostrarla directamente por inducción matemática: recuerde que suele ser más fácil demostrar un teorema fuerte que uno más débil. Por tanto, vamos a suponer que existe un número real $x > 1$ tal que $f_n \geq x^n$ para todo entero n suficientemente grande (esta afirmación no puede en modo alguno ser cierta para *todo* entero positivo n, puesto que evidentemente falla para $n \leq 2$). En símbolos:

$$\textit{Conjetura}: \qquad (\exists x > 1)(\overset{\infty}{\forall} n \in \mathbb{N})[f_n \geq x^n]$$

En el teorema que deseamos demostrar hay dos incógnitas: el valor de x y el significado preciso de «para todo n suficientemente grande». Ignoremos esto último por el momento. Supongamos que $P_x(n)$ denota «$f_n \geq x^n$». Considérese cualquier entero n suficientemente grande. El enfoque de inducción constructiva consiste en preguntarnos para qué valores de x se sigue $P_x(n)$ *tomando como*

base la hipótesis de inducción especificada parcialmente y consistente en que $P_x(m)$ es válido para todo entero m que sea menor que n pero que tenga un valor suficientemente grande. Empleando la definición de la sucesión de Fibonacci y esta hipótesis, y siempre y cuando $n-1$ y $n-2$ sean también «suficientemente grandes»:

$$f_n = f_{n-1} + f_{n-2} \geq x^{n-1} + x^{n-2} = (x^{-1} + x^{-2})x^n$$

Para concluir que $f_n \geq x^n$, necesitamos que sea $x^{-1} + x^{-2} \geq 1$, o equivalentemente que $x^2 - x - 1 \leq 0$. Por álgebra elemental, puesto que sólo estamos interesados en el caso $x > 1$, la resolución de esta ecuación implica que $1 < x \leq \phi = (1+\sqrt{5})/2$.

Hemos determinado que $P_x(n)$ se sigue de $P_x(n-1)$ y de $P_x(n-2)$ siempre y cuando $1 \leq x \leq \phi$. Esto se corresponde con demostrar el paso de inducción en una demostración por inducción matemática. Para aplicar el principio de inducción matemática, y concluir que la sucesión de Fibonacci crece con rapidez exponencial, también hay que resolver la base. En este caso, dado que la veracidad de $P_x(n)$ depende también de la de $P_x(n-1)$ y de $P_x(n-2)$, basta con demostrar que la propiedad P_x es válida para dos valores positivos consecutivos para afirmar que es válida desde ese momento en adelante.

Resulta que no hay enteros n tales que $f_n \geq \phi^n$. Sin embargo, es fácil hallar dos enteros consecutivos para los cuales la propiedad P sea válida para todo x estrictamente menor que ϕ. Por ejemplo, tanto $P_x(11)$ como $P_x(12)$ son válidos cuando $x = 3/2$. Por tanto, $fm \geq \left(\frac{3}{2}\right)^n$ para todo $n \geq 11$. Esto completa la demostración de que la sucesión de Fibonacci crece *al menos* exponencialmente. Se puede utilizar el mismo proceso para demostrar que no crece *más deprisa* que exponencialmente: $f_n \leq y^n$ para todo entero positivo n siempre y cuando sea $y \geq \phi$. Una vez más, la condición sobre y no sale de la nada: se obtiene por inducción constructiva cuando se intenta obtener cotas de y que hagan que el paso de inducción sea verdadero. Reuniendo todas estas observaciones, concluimos que f_n crece exponencialmente; más exactamente, crece como una potencia de un número próximo a ϕ. Lo sorprendente es que podamos alcanzar esta conclusión sin necesidad de una fórmula explícita como la de Moivre.

Nuestro segundo ejemplo de inducción constructiva concierne al análisis del algoritmo evidente para calcular la sucesión de Fibonacci:

```
función Fibonacci(n)
  si n < 2 entonces devolver n
  sino devolver Fibonacci(n − 1) + Fibonacci(n − 2)          (*)
```

Supongamos que $g(n)$ representa el número de veces que se ejecuta la instrucción (*) cuando se invoca *Fibonacci* (n) (contando las instrucciones efectuadas en llamadas recursivas). Esta función es interesante porque $g(n)$ proporciona una cota del tiempo requerido por una llamada a *Fibonacci* (n).

Claramente, $g(0) = g(1) = 0$. Cuando $n \leq 2$, la instrucción (*) se ejecuta una vez en el nivel superior, y $g(n-1)$ y $g(n-2)$ veces en la primera y segunda llamada recursiva, respectivamente. Por tanto,

$$\begin{cases} g(0) = g(1) = 0 & \text{y} \\ g(n) = g(n-1) + g(n-2) + 1 & \text{para } n \geq 2 \end{cases}$$

Esta fórmula es similar a la recurrencia que define la propia sucesión de Fibonacci en sí. Por tanto, parece razonable conjeturar la existencia de constantes reales y positivas a y b tales que $af_n \leq g(n) \leq bf_n$ para todo entero n suficientemente grande. Empleando la inducción constructiva, es sencillo encontrar que $af_n \leq g(n)$ es válido para todo n suficientemente grande siempre y cuando sea válido para dos enteros consecutivos, independientemente del valor de a. Por ejemplo, haciendo $a = 1$, $f_n \leq g(n)$ es válido para todo $n \geq 2$.

Sin embargo, cuando se intenta demostrar la otra parte de nuestra conjetura, a saber, que existe un b tal que $g(n) \leq bf_n$ para todo n suficientemente grande, nos encontramos con dificultades. Para ver lo que sucede, supongamos que $P_b(n)$ denota «$g(n) \leq bf_n$», y consideremos cualquier entero n suficientemente grande (que ya precisaremos más adelante). Deseamos determinar las condiciones aplicables al valor de b que hacen que $P_b(n)$ se siga de la hipótesis consistente en que $P_b(m)$ es válido para todo $m < n$ suficientemente grande. Empleando la definición de la sucesión de Fibonacci y esta hipótesis de inducción especificada parcialmente, y siempre y cuando $n - 1$ y $n - 2$ sean también suficientemente grandes, $g(n) = g(n-1) + g(n-2) + 1 \leq bf_{n-1} + bf_{n-2} + 1 = bf_n + 1$ en donde la última igualdad proviene de $f_n = f_{n-1} + f_{n-2}$. De esta manera, inferimos que $g(n) \leq bf_n + 1$, pero no que $g(n) \leq bf_n$. Independientemente del valor de b ¡no conseguimos que funcione el paso de inducción!

¿Significa esto que la conjetura original era falsa, o es meramente que la inducción constructiva es impotente para demostrarla? La respuesta es: ninguna de las dos. El truco consiste en utilizar la inducción constructiva para demostrar que existen constantes reales positivas b y c tales que $g(n) \leq bf_n - c$ para todo n suficientemente grande. Esto puede parecer extraño, puesto que $g(n) \leq bf_n - c$ es una afirmación más fuerte que $g(n) \leq bf_n$, que no éramos capaces de demostrar. Podemos tener esperanzas de éxito, sin embargo, basándonos en que si la afirmación que hay que demostrar es más fuerte, también lo es la hipótesis de inducción que nos permite utilizar; véase el final de la Sección 1.6.1.

Considere cualquier entero n suficientemente grande. Tenemos que determinar para qué valores de b y c se sigue la veracidad de $g(n) \leq bf_n - c$, a partir de la hipótesis de inducción parcialmente especificada y consistente en que $g(m) \leq bf_m - c$ para todo $m < n$ suficientemente grande. Utilizando la definición de la sucesión de Fibonacci y esta hipótesis, y siempre y cuando $n - 1$ y $n - 2$ sean también suficientemente grandes:

$$\begin{aligned} g(n) &= g(n-1) + g(n-2) + 1 \\ &\leq bf_{n-1} - c + bf_{n-2} - c + 1 = bf_n - 2c + 1 \end{aligned}$$

Para concluir que $g(n) \leq bf_n - c$, basta que sea $-2c + 1 \leq -c$, esto es, que $c \leq 1$. Por tanto hemos demostrado que la veracidad de nuestra conjetura acerca de cualquier entero n dado se sigue de suponer su veracidad para los dos enteros anteriores, siempre y cuando $c \leq 1$, e independientemente del valor de b. Antes de poder afirmar el teorema deseado, seguimos necesitando determinar los valores de b y c que hacen que sea válida para dos enteros consecutivos. Por ejemplo, $b = 2$ y $c = 1$ hacen que funcione para $n = 1$ y $n = 2$, y por tanto $g(n) \leq 2f_n - 1$ para todo $n \geq 1$.

La idea clave de fortalecer la afirmación especificada de forma incompleta que hay que probar cuando falla la inducción constructiva puede haber aparecido, una vez más, igual que un conejo saliendo de un sombrero. Sin embargo, la idea aparece de forma muy natural con la experiencia. Para adquirir esta experiencia, haga los problemas 1.31 y 1.33. A diferencia de los ejemplos de Fibonacci, que se podrían haber resuelto fácilmente mediante las técnicas de la Sección 4.7, los casos que se atacan en estos problemas se resuelven del mejor modo posible mediante inducción constructiva.

1.7 RECORDATORIOS

En esta sección recordamos al lector algunos resultados elementales acerca de límites, sumas de series sencillas, combinatoria y probabilidad. Los capítulos posteriores utilizarán las proposiciones que se presentan aquí. Nuestra presentación es sucinta, y se omiten casi todas las demostraciones, puesto que esperamos que la mayoría de los lectores haya estudiado ya este material.

1.7.1 Límites

Sea $f(n)$ cualquier función de n. Diremos que $f(n)$ tiende a un límite a cuando n tiende a infinito si $f(n)$ es casi igual a a cuando n es grande. La siguiente definición formal hace más precisa esta noción.

Definición 1.7.1. *Se dice que la función $f(n)$ tiende al límite* a *cuando n tiende a infinito si para todo número real positivo δ, independientemente de lo pequeño que sea, $f(n)$ dista de* a *en una cantidad menor que δ para todos los valores de n suficientemente grandes*

En otras palabras, independientemente de lo pequeño que sea el número δ, podemos hallar un umbral $n_0(\delta)$ correspondiente a δ tal que $f(n)$ dista de a en menos que δ para todos los valores de n mayores o iguales que $n_0(\delta)$. Si $f(n)$ tiende al límite a cuando n tiende a infinito, escribimos:

$$\lim_{n \to \infty} f(n) = a$$

Hay muchas funciones, desde luego, que no tienden a un límite cuando n tiende a infinito. La función n^2, por ejemplo, puede hacerse tan grande como sea preciso sin más que seleccionar un valor de n suficientemente grande. Se dice que estas funciones tienden a infinito cuando n tiende a infinito. La definición formal es como sigue.

Definición 1.7.2. *Se dice que la función $f(n)$ tiende a $+\infty$ si para todo número Δ, independientemente de lo grande que sea, $f(n)$ es mayor que Δ para todos los valores de n suficientemente grandes*

Una vez más, esto significa que podemos hallar un umbral $n_0(\Delta)$ correspondiente a Δ, tal que $f(n)$ es mayor que Δ para todos los valores de n mayores o iguales que $n_0(\Delta)$. Escribiremos:

$$\lim_{n\to\infty} f(n) = +\infty$$

Hay una definición parecida para fórmulas tales como $-n^2$, que pueden tomar valores negativos cada vez más grandes a medida que n tiende a infinito. Se dice que estas funciones tienden a menos infinito.

Por último, cuando $f(n)$ no tiende a un límite, ni tampoco a $+\infty$ o $-\infty$, diremos que $f(n)$ *oscila* cuando n tiende a infinito. Si es posible encontrar una constante positiva K tal que $-K < f(n) < K$ para todos los valores de n, diremos que $f(n)$ oscila de *forma finita*; en caso contrario, $f(n)$ oscila de *forma infinita*. Por ejemplo, la función $(-1)^n$ oscila de forma finita; la función $(-1)^n n$ oscila de forma infinita.

Las proposiciones siguientes enuncian algunas de las propiedades generales de los límites.

Proposición 1.7.3. *Si dos funciones $f(n)$ y $g(n)$ tienden respectivamente a los límites a y b cuando* n *tiende a infinito, entonces $f(n) + g(n)$ tiende al límite $a+b$*

Proposición 1.7.4. *Si dos funciones $f(n)$ y $g(n)$ tienden respectivamente a los límites a y b cuando* n *tiende a infinito, entonces $f(n)g(n)$ tiende al límite ab*

Estas dos proposiciones se pueden extender a la suma o producto de cualquier número finito de funciones de n. Un caso particular importante de la proposición 1.7.4 es aquel en el cual $g(n)$ es constante. La proposición afirma, entonces, que si

el límite de $f(n)$ es a, entonces el límite de $cf(n)$ es ca, en donde c es cualquier constante. Es perfectamente posible que $f(n) + g(n)$ o bien $f(n)g(n)$ tiendan a un límite aun cuando ni $f(n)$ ni $g(n)$ tengan límite; véase el problema 1.34. Por último, la proposición siguiente afecta a la división.

Proposición 1.7.5. *Si dos funciones f(n) y g(n) tienden respectivamente a los límites a y b cuando* n *tiende a infinito, y b no es cero, entonces f(n)/g(n) tiende al límite* a/b

Estas proposiciones, aunque sean sencillas, resultan sorprendentemente potentes. Por ejemplo, supongamos que se desea conocer el comportamiento de la función racional más general de n, a saber:

$$S(n) = \frac{a_0 n^p + a_1 n^{p-1} + \ldots + a_p}{b_0 n^q + b_1 n^{q-1} + \ldots + b_q}$$

en donde ni a_0 ni b_0 son cero. Si escribimos $S(n)$ en la forma

$$S(n) = n^{p-q}\left\{\left(a_p + \frac{a_1}{n} + \ldots + \frac{a_p}{n^p}\right)\Big/\left(b_0 + \frac{b_1}{n} + \ldots + \frac{b_q}{n^q}\right)\right\}$$

y aplicando las proposiciones anteriores, resulta sencillo ver que la función que va entre llaves tiende al limite a_0/b_0 cuando n tiende a infinito. Además, n^{p-q} tiende al límite 0 si $p < q$; $n^{p-q} = 1$ y además n^{p-q} tiende al límite 1 si $p = q$; y n^{p-q} tiende al infinito si $p>q$. Por tanto:

$$\lim_{n\to\infty} S(n) = 0 \quad \text{cuando } p < q$$

$$\lim_{n\to\infty} S(n) = a_0 / b_0 \quad \text{cuando } p < q$$

y $S(n)$ tiende a más o menos infinito cuando $p > q$ dependiendo del signo de a_0/b_0.

Proposición 1.7.6. *Si* $\lim_{n\to\infty} (f(n) + 1)/f(n)) = a$, $-1 < a < 1$, *entonces* $\lim_{n\to\infty} f(n) = 0$. *Si f(n) es positivo y* $\lim_{n\to\infty} (f(n+1) / f(n)) = a > 1$, *entonces f(n) tiende a infinito.*

Esta proposición se puede utilizar para determinar el comportamiento cuando n tiende a infinito de $f(n) = n^r x^n$, donde r es cualquier entero positivo. Si $x = 0$ entonces $f(n) = 0$ para todos los valores de n. En caso contrario

$$\frac{f(n+1)}{f(n)} = \left(\frac{n+1}{n}\right)^r x$$

tiende a x cuando n tiende a infinito. (Utilícense las proposiciones anteriores). Por tanto si $-1 < x < 1$ entonces $f(n)$ tiende a 0, y si $x > 1$ entonces $f(n)$ tiende a infinito. Si $x = 1$ entonces $f(n) = n^r$, lo cual tiende claramente a infinito. Por último es fácil ver que si $x \leq -1$ entonces $f(n)$ presenta una oscilación infinita. El problema 1.35 muestra que el comportamiento de $n^{-r} x^n$ es el mismo, salvo que $f(n)$ tiende a 0 cuando $x = 1$ o $x = -1$.

La *regla de l'Hôpital* se puede utilizar en algunas ocasiones cuando resulta imposible aplicar la proposición 1.7.5. Una forma sencilla de esta regla es la siguiente:

Proposición 1.7.7. *(De l'Hôpital) Supóngase que*

$$\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = 0,$$

o alternativamente que estos dos límites son infinitos. Supóngase además que los dominios de f y g se pueden extender hasta algún intervalo real $[n_0, +\infty)$ *de tal forma que las nuevas funciones correspondientes* $\hat{f}$ *y* $\hat{g}$ *son diferenciables en este intervalo, y además que* $\hat{g}'(x)$*, la derivada de* $\hat{g}(x)$*, nunca es cero para* $x \in [n_0, +\infty)$*, entonces*

$$\lim_{n\to\infty} f(n)/g(n) = \lim_{x\to\infty} \hat{f}'(x)/\hat{g}'(x)$$

Como ejemplo sencillo, supongamos que $f(n) = \log n$ y $g(n) = n^a$, en donde $a > 0$ es una constante positiva arbitraria. Ahora bien, tanto $f(n)$ como $g(n)$ tienden a infinito cuando n tiende a infinito, así que no se puede utilizar la proposición 1.7.5. Sin embargo, si extendemos $f(n)$ a $\hat{f}(x) = \log x$ y $g(n)$ a $\hat{g}(x) = x^a$, entonces la regla de l'Hôpital nos permite concluir que

$$\lim_{n\to\infty} \log n / n^a = \lim_{x\to\infty} (1/x)/(ax^{a-1}) = \lim_{x\to\infty} 1/(ax^a) = 0$$

sea cual fuere el valor positivo de a.

Por último, la proposición siguiente es útil en algunas ocasiones aunque es muy sencilla de demostrar.

Proposición 1.7.8. *Si las dos funciones* $f(n)$ *y* $g(n)$ *tienden a los límites a y b respectivamente cuando n tiende a infinito, y si* $f(n) \leq g(n)$ *para todo n suficientemente grande, entonces* $a \leq b$.

1.7.2 Series sencillas

Supongamos que $u(n)$ es cualquier función de n definida para todos los valores de n. Si sumamos los valores de $u(i)$ para $i = 1,2, ..., n$, se obtiene otra función de n, a saber:

$$s(n) = u\,(1) + u\,(2) + \ldots + u\,(n)$$

Suele ser conveniente cambiar un poco la notación, y escribir esta ecuación en la forma

$$s_n = u_1 + u_2 + \ldots + u_n$$

o bien simplemente

$$s_n = \sum_{i=1}^{n} u_i$$

que leeremos como «la suma de u_i cuando i va desde 1 hasta n ». Ahora si s_n tiende a un límite s cuando n tiende a infinito, se tiene que

$$s = \lim_{n\to\infty} \sum_{i=1}^{n} u_i$$

lo cual se suele escribir como

$$s = \sum_{i=1}^{\infty} u_i$$

o bien como

$$s = u_1 + u_2 + \ldots$$

en donde los puntos suspensivos denotan que la serie continúa de forma indefinida. En este caso diremos que la serie es *convergente*, y llamaremos a s la *suma* de la serie.

Si por otra parte s_n no tiende a un límite, sino que s_n tiende a $+\infty$ o a $-\infty$, entonces diremos que la serie *diverge*, a $+\infty$ o quizá a $-\infty$. Finalmente, si s_n no tiende a un límite, ni a $+\infty$ ni a $-\infty$, entonces diremos que la serie oscila (de forma finita o quizá infinita). Es evidente que una serie que no tenga términos negativos debe de converger, o bien diverger, a $+\infty$: no podrá oscilar.

Dos clases de series especialmente sencillas son las *series aritméticas* y las *series geométricas*. En una serie aritmética la diferencia entre términos sucesivos es

constante, así que se pueden representar los n primeros términos de la serie en la forma

$$a, a + d, a + 2d, \ldots, a + (n - 1)\, d$$

en donde a, el primer término, y d, la diferencia entre términos sucesivos, son constantes adecuadas. En una serie geométrica, la razón de términos sucesivos es constante, de tal forma que aquí los n primeros términos de la serie se pueden representar en la forma

$$a, ar, ar^2, \ldots, ar^{n-1}$$

Las proposiciones siguientes dan la suma de estas series.

Proposición 1.7.9. (Serie aritmética) *Sea* s_n *la suma de los* n *primeros términos de la serie aritmética* $a, a + d, a + 2d, \ldots$ *Entonces* $s_n = an + n\,(n - 1)\, d/2$

La serie diverge a no ser que $a = d = 0$, en cuyo caso $s_n = 0$ para todo n. La proposición se demuestra fácilmente. En primer lugar se escribe la suma en la forma

$$s_n = a + (a + d) + \ldots + (a + (n - 2)d) + (a + (n - 1)d)$$

y después se escribe de nuevo en la forma

$$s_n = (a + (n - 1)d) + (a + (n - 2)d) + \ldots + (a + d) + a$$

Al sumar los términos correspondientes de estas dos ecuaciones, se obtiene

$$2s_n = (2a + (n - 1)d) + (2a + (n - 1)d) + \ldots + (2a + (n - 1)d) + (2a + (n - 1)d)$$

en donde hay n términos iguales en el lado derecho. El resultado se sigue inmediatamente.

Proposición 1.7.10. (Serie geométrica) *Sea* s_n *la suma de los* n *primeros términos de la serie geométrica* $a, ar, ar^2, \ldots$ *Entonces* $s_n = a\,(1 - r^n) / (1 - r)$, *salvo en el caso especial en el cual* $r = 1$, *en donde* $s_n = an$

Para ver esto, se escribe

$$s_n = a\,(1 + r + r^2 + \ldots + r^{n-1})$$

de tal manera que

$$rs_n = a\,(r + r^2 + r^3 + \ldots + r^n)$$

Al restar la segunda ecuación de la primera, se obtiene inmediatamente

$$(1 - r)\,s_n = a\,(1 - r^n)$$

En el caso general (esto es, cuando $r \neq 1$) la suma s_n de una serie geométrica tiende a un límite si y sólo si r^n también lo hace. Esto nos da la proposición siguiente.

Proposición 1.7.11. (Serie geométrica infinita) *La serie geométrica infinita* $a + ar + ar^2 + \ldots$. *es convergente y tiene como suma* $a / (1 - r)$ *si y sólo si* $-1 < r < 1$.

Se puede utilizar una técnica similar para obtener un resultado útil que concierne a otra serie más. Si escribimos

$$s_n = r + 2r^2 + 3r^3 + \ldots + (n - 1)\,r^{n-1}$$

tenemos que

$$rs_n = r^2 + 2r^3 + 3r^4 + \ldots (n - 1)\,r^n$$

Restando la segunda ecuación de la primera, se obtiene

$$(1 - r)\,s_n = r + r^2 + r^3 + \ldots + r^{n-1} - (n - 1)\,r^n$$

$$= r\,(1 + r + r^2 + \ldots + r^{n-1})\; nr^n$$

$$= [r\,(1 - r^n) \,/\, (1 - r)] - nr^n$$

en virtud de la proposición 1. 7. 10. Por tanto:

$$S_n = r + 2r^2 + \ldots + (n-1)r^{n-1} = \frac{r(1-r)^n}{(1-r)^2} - \frac{nr^n}{(1-r)}$$

El lado derecho tiende a un límite cuando n tiende a infinito si $-1 < r < 1$ (utilizando la proposición 1.7.6), lo cual nos da el resultado siguiente.

Proposición 1.7.12. *La serie infinita* $r + 2r^2 + 3r^3 + \ldots$ *converge cuando* $-1 < r < 1$, *y en este caso su suma es* $r/(1 - r)^2$.

A continuación acudiremos a las series de la forma $1^r, 2^r, 3^r, \ldots$, en donde r es un entero positivo. Cuando $r = 0$ la serie es simplemente $1, 1, 1, \ldots$, y obtenemos el resultado trivial $\Sigma_{i=1}^{n} 1 = n$. La proposición siguiente nos ofrece una forma general para manejar los casos en que $r > 0$.

Proposición 1.7.13. *Para cualquier entero $k \geq 0$ tenemos*

$$\sum_{i=1}^{n} i(i+1) \ldots (i+k) = n\,(n+1) \ldots (n+k+1) \,/\, (k+2)$$

Esta proposición se demuestra fácilmente por inducción matemática sobre n. Obsérvese que *no* es necesario utilizar inducción sobre k, que no pasa de ser un parámetro en la fórmula.

Utilizando la proposición 1.7.13 es fácil obtener las sumas de las distintas series de interés.

Proposición 1.7.14. $$\sum_{i=1}^{n} i = n\,(n+1) \,/\, 2$$

Este resultado es simplemente la proposición 1.7.13 con $k = 0$.

Proposición 1.7.15. $$\sum_{i=1}^{n} i^2 = n\,(n+1)\,(2n+1) \,/\, 6$$

Para demostrar esto, tomemos

$$\begin{aligned}
\sum_{i=1}^{n} i^2 &= \sum_{i=1}^{n} (i\,(i+1) - i) \\
&= \sum_{i=1}^{n} i\,(i+1) - \sum_{i=1}^{n} i \\
&= n\,(n+1)\,(n+2) \,/\, 3 - n\,(n+1) \,/\, 2 \\
&= n\,(n+1)\,(2n+1) \,/\, 6
\end{aligned}$$

en donde hemos utilizado la proposición 1.7.13 dos veces para evaluar las dos sumas.

Una línea similar de ataque se podría utilizar para evaluar cualquier otra serie de este tipo. De hecho, tal como le mostramos a continuación, resulta sencillo probar la siguiente proposición general.

Proposición 1.7.16. *Sea r cualquier entero positivo. Entonces*

$$\sum_{i=1}^{n} i^r = n^{r+1} / (r+1) + p_r(n)$$

en donde $p_r(n)$ es un polinomio de grado no mayor que r

El esbozo de la demostración es como sigue:

$$\begin{aligned}\sum_{i=1}^{n} i^r &= \sum_{i=1}^{n} i\,(i+1)\ldots(i+r-1) + \sum_{i=1}^{n} p\,(i) \\ &= n\,(n+1)\ldots(n+r)/(r+1) + p'(n) \\ &= n^{r+1} / (r+1) + p''(n)\end{aligned}$$

en donde $p(i)$ es un polinomio de grado no mayor que $r-1$, y $p'(n)$ y $p''(n)$ son polinomios de grado no mayor que r. Dejaremos al lector rellenar los detalles del argumento.

Finalmente consideraremos de manera breve las series de la forma $1^{-r}, 2^{-r},\ldots$ donde r es un entero positivo. Cuando $r = 1$ obtenemos la serie $1, 1/2, 1/3,\ldots$ que se conoce con el nombre de *serie armónica*. Es fácil mostrar que esta serie diverge. La proposición siguiente nos proporciona una mejor idea de su comportamiento.

Proposición 1.7.17. *(Serie armónica) Sea H_n la suma de los* n *primeros términos de la serie armónica* $1, 1/2, 1/3,\ldots$ *Entonces* $\log(n+1) < H_n \leq 1 + \log n$

Para demostrarla considérese la figura 1.8. El área bajo la «escalera» nos da la suma de la serie armónica; el área por encima de la curva inferior, $y = 1/(x+1)$, es menor que esta suma, mientras que el área bajo la curva superior, que es $y = 1$ para $x < 1$ e $y = 1/x$ en lo sucesivo, es mayor o igual a la suma. Por tanto

$$\int_0^n \frac{dx}{x+1} < H_n \leq 1 + \int_1^n \frac{dx}{x}$$

de lo cual se sigue inmediatamente la proposición. Una estimación más precisa de H_n para n grande se puede obtener a partir de

$$\lim_{n\to\infty}\left(1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}-\log n\right)=\gamma$$

donde $\gamma \approx 0.57721\ldots$ que es la *constante de Euler,* pero la demostración va más allá del alcance de nuestro libro.

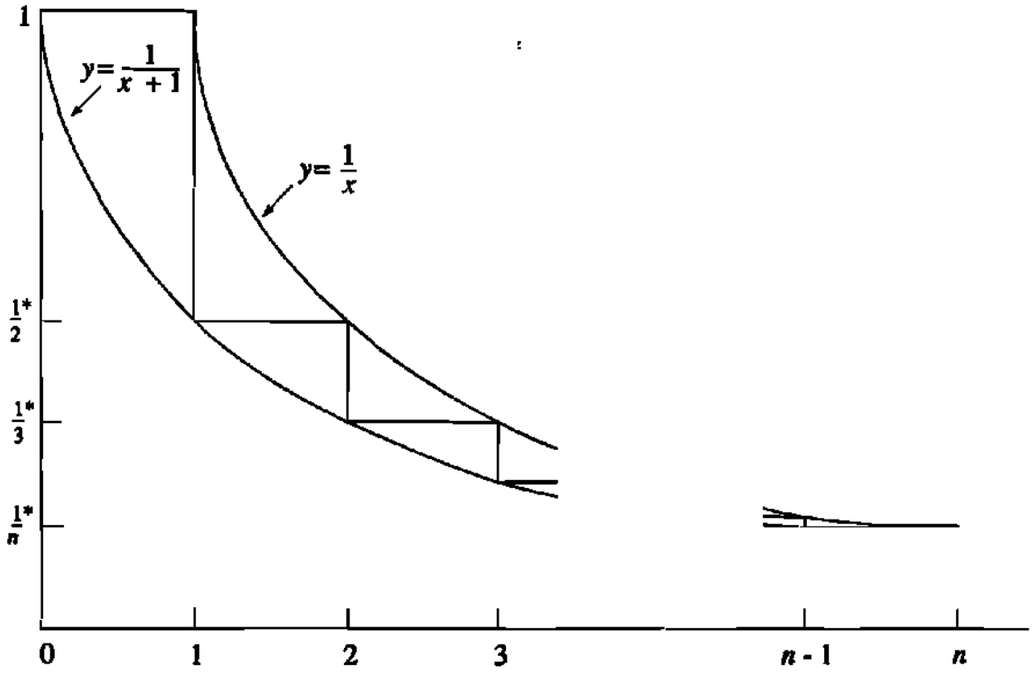


Figura 1.8 Suma de la series armónicas

Resulta sencillo mostrar que las series de la forma $1, 1/2^r, 1/3^r,\ldots$ cuando $r > 1$ son todas convergentes, y que la suma de todas estas series es menor que $r/(r-1)$; véase el problema 1.39. Por ejemplo:

$$\lim_{n\to\infty}\left(1+\frac{1}{2^2}+\frac{1}{3^2}+\ldots+\frac{1}{n^2}\right)=\frac{\pi^2}{6}\approx 1{,}64493$$

Sin embargo no resulta fácil calcular el valor exacto de estas sumas.

1.7.3 Combinatoria básica

Supongamos que se tienen n objetos que son suficientemente distintos para que podamos reconocer cada uno: diremos que estos objetos son *distinguibles*. Para facilitar la exposición supongamos que están rotulados con las letras a, b, y así sucesivamente, teniendo cada objeto un rótulo distinto. De ahora en adelante aludiremos simplemente a a, por ejemplo, cuando queramos decir «el objeto cuyo rótulo es a».

Nuestra primera definición concierne al número de formas en que se pueden ordenar estos n objetos.

Definición 1.7.18. *Una permutación de n objetos es una disposición ordenada de los objetos.*

Por ejemplo, si tenemos cuatro objetos a, b, c y d, podemos disponerlos por orden de 24 maneras distintas:

abcd abdc acbd acdb adbc adcb
bacd badc bcad bcda bdac bdca
cabd cadb cbad cbda cdab cdba
dabc dacb dbac dbca dcab dcba

El primer objeto de la permutación se puede seleccionar de n maneras; una vez que se ha seleccionado el objeto, el segundo se puede seleccionar de $n - 1$ formas distintas; una vez que se han seleccionado el primero y el segundo, el tercero se puede seleccionar de $n - 2$ formas distintas, y así sucesivamente. Hay dos posibilidades cuando se ha seleccionado el penúltimo objeto, y solamente hay una forma de seleccionar el último objeto. El número total de permutaciones de n objetos es por tanto

$$n\,(n-1)\,(n-2)\,\ldots\,2:1 = n!$$

A continuación consideraremos el número de formas de seleccionar un cierto número de estos objetos, independientemente del orden en el cual hagamos nuestras selecciones.

Definición 1.7.19. *Una combinación de r objetos de entre n objetos es una selección de r objetos sin tener en cuenta el orden.*

Por ejemplo, si tenemos cinco objetos a, b, c, d y e, podemos seleccionar tres de ellos de 10 maneras distintas si no se tiene en cuenta el orden:

abc abd abe acd ace
ade bcd bce bde cde

Una selección tal como *eba* no aparece en esta lista, puesto que es lo mismo que *abe* cuando no se tiene en cuenta el orden de los objetos.

Cuando hacemos nuestra selección de r objetos de entre n, hay n formas de efectuar la primera opción. Cuando se ha seleccionado el primer objeto, quedan $n - 1$ formas para seleccionar el segundo, y así sucesivamente. Cuando se selecciona el último de los r objetos deseados, quedan $n - r + 1$ posibilidades. Por tanto hay

$$n\,(n-1)\,(n-2)\,\ldots\,(n-r+1)$$

formas de seleccionar r objetos de n cuando *se tiene* en cuenta el orden. Sin embargo cuando *no se tiene* en cuenta el orden, se pueden permutar los r objetos seleccionados de cualquier forma que queramos, y sigue contando como la misma combinación. En el ejemplo anterior, las seis opciones ordenadas *abc, acb, bac, bca, cab* y *cba* cuentan todas ellas como una misma combinación. Puesto que hay $r!$ formas de permutar r objetos, el número de formas de seleccionar r objetos de entre n cuando no se tiene en cuenta el orden es

$$\binom{n}{r} = \frac{n(n-1)(n-2)\ldots(n-r+1)}{r!} \tag{1.1}$$

Hay varias notaciones alternativas que se utilizan para el número de combinaciones de n objetos tomados r a r: entre otras, se puede encontrar ${}_nC_r$ y C_r^n. Esto da cuenta del hábito frecuente pero ilógico consistente en escribir $\binom{n}{r}$ pero leyéndolo en la forma « n C r ».
Cuando $r > n$ la ecuación 1.1 da $\binom{n}{r} = 0$, lo cual tiene sentido: no hay forma de seleccionar más de n objetos de entre n. Resulta conveniente tomar $\binom{n}{0} = 1$ (porque hay sólo una forma de no seleccionar ningún objeto), y cuando $r < 0$, que no tiene significado combinatorio, definimos $\binom{n}{r} = 0$. Cuando $0 \le r \le n$, la ecuación 1.1 se puede escribir de forma cómoda como

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Un argumento sencillo nos permite obtener una relación importante. Considérese cualquiera de los n objetos, y dígase que este objeto es «especial». Ahora, cuando seleccionemos r objetos de entre los n disponibles, podemos distinguir aquellas selecciones que incluyan al objeto especial y aquellas que no. Por ejemplo, si tenemos que seleccionar tres objetos de entre *a, b, c, d* y e, y el objeto especial es *b*, entonces las selecciones que incluyen al objeto especial son *abc, abd, abe, bcd, bce,* y *bcd,* mientras que las que no incluyen al objeto especial son *acd, ace, ade* y *cde.* Para hacer una selección del primer tipo, podemos seleccionar primero el objeto especial (puesto que el orden de nuestras selecciones no es importante), y completar entonces nuestra selección escogiendo $r - 1$ objetos de entre los $n - 1$ que quedan; esto se puede hacer de $\binom{n-1}{r-1}$ maneras. Para hacer una selección de la segunda clase, debemos de seleccionar nuestros r objetos de entre los $n - 1$ que no son especiales; esto se puede hacer de $\binom{n-1}{r}$ maneras. Puesto que la selección de r objetos de entre los n disponibles es de una clase o de otra, es preciso que se cumpla

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}, \quad 1 \le r \le n.$$

Esta fórmula también se puede demostrar utilizando la ecuación 1.1.

La fórmula nos da una forma sencilla de tabular los valores de $\binom{n}{r}$, según se ilustra en la figura 1. 9.

Aquí cada figura de las tablas se puede calcular fácilmente a partir de los elementos de la fila precedente. Una tabla de este tipo que muestra los valores $\binom{n}{r}$ suele denominarse *Triángulo de Pascal.*

Los valores $\binom{n}{r}$ también se conocen con el nombre de *coeficientes binomiales* como consecuencia del papel que desempeñan en el teorema siguiente. El teorema, que no intentaremos demostrar, se suele atribuir a Newton, aunque parece haber sido conocido ya por Omar Khayyam unos 600 años antes.

n/r	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Figura 1.9 Combinaciones de n objetos tomados r a r

Teorema 1.7. 20 (Newton). *Sea* n *un entero positivo. Entonces*

$$(1+x)^n = 1+\binom{n}{1}x+\binom{n}{2}x^2+\ldots+\binom{n}{n-1}x^{n-1}+x^n$$

Utilizando este teorema es sencillo obtener resultados interesantes concernientes a los coeficientes binomiales. Por ejemplo, al hacer $x = 1$ se obtiene inmediatamente

$$\sum_{r=0}^{n}\binom{n}{r} = 2^n$$

En términos combinatorios, la suma de la izquierda es el número de formas de seleccionar un número arbitrario de objetos (incluyendo 0 objetos) de entre n, en donde no se tiene en cuenta el orden. Puesto que hay dos posibilidades para cada uno de los n objetos, se puede seleccionar o bien se puede descartar, entonces hay 2^n maneras de hacer esto. De forma similar, al hacer $x = -1$ en el teorema 1.7.20 se encuentra que

$$\sum_{\substack{r=0 \\ r \text{ impar}}}^{n}\binom{n}{r} = \sum_{\substack{r=0 \\ r \text{ par}}}^{n}\binom{n}{r}.$$

1.7.4 Probabilidad elemental

La teoría de la probabilidad concierne al estudio de fenómenos aleatorios; esto es, de aquellos fenómenos cuyo futuro no es predecible con certeza. Para aplicar la teoría de la probabilidad, consideraremos el fenómeno de interés como un *experimento aleatorio* cuyo resultado no se conoce con certeza por anticipado. Tal experimento podría consistir, por ejemplo, en tirar un dado e ir apuntando la cara que queda hacia arriba, o bien podría consistir en contar cuántos coches van pasando por un cierto punto en un periodo determinado de tiempo, o bien en medir el tiempo de respuesta de un sistema de computadoras. El dato anotado para cada experimento se denomina su *resultado.*

El conjunto de todas los posibles resultados de un experimento aleatorio se denomina *espacio muestral* del experimento. En los párrafos siguientes notaremos este espacio muestral mediante S. Los resultados individuales se denominan *puntos muestrales* o *sucesos elementales.* Por ejemplo, cuando tiramos un dado ordinario, hay seis resultados posibles, a saber, los valores del 1 al 6. Para este experimento, entonces, el espacio muestral es $S = \{1, 2, 3, 4, 5, 6\}$. Para el experimento aleatorio que consiste en contar los coches que pasan por un cierto punto, el espacio muestral es $S = \{0, 1, 2,\ldots\}$. Para el experimento aleatorio que consiste en medir el tiempo de respuesta de un sistema de computadoras, el espacio muestral es $S = \{t \mid t > 0\}$.

Un espacio muestral puede ser finito o infinito, y puede ser discreto o continuo. Se dice que un espacio muestral es discreto si el número de puntos muestrales es finito, o bien si se pueden rotular en la forma 1, 2, 3, y así sucesivamente utilizando los números enteros; en caso contrario, el espacio muestral es continuo. En los ejemplos anteriores, S es finito y por tanto discreto para el experimento aleatorio consistente en tirar un dado; S es infinito pero discreto para el experimento consistente en contar coches, porque los resultados posibles se pueden hacer corresponder con los enteros positivos; y S es continuo para el experimento consistente en medir un tiempo de respuesta. En este libro nos ocuparemos casi por entero de experimentos aleatorios cuyo espacio muestral es finito, y por tanto discreto.

Se define ahora un *suceso* como una colección de puntos muestrales, esto es, como un subconjunto del espacio muestral. Se dice que un suceso A *sucede* si el experimento aleatorio se lleva a cabo y el resultado observado es un elemento del conjunto A. Por ejemplo, cuando tiramos un dado, el suceso descrito por la frase «el número mostrado en el dado es impar» se corresponde con el subconjunto $A = \{1, 3, 5\}$ del espacio muestral. Cuando contamos coches el suceso descrito por «el número observado de coches es menor que 20» se corresponde con el subconjunto $A = \{0, 1, 2,\ldots, 18, 19\}$, y así sucesivamente. Informalmente, utilizaremos la palabra «suceso» para aludir bien a la frase que lo describe, o al subconjunto correspondiente del espacio muestral. En particular, todo el espacio muestral S es un suceso conocido con el nombre de *suceso universal,* y el conjunto vacío $\emptyset$ es un suceso denominado el *suceso imposible.*

Puesto que un espacio muestral S es un conjunto y un suceso A es un subconjunto de S, podemos formar nuevos sucesos mediante las operaciones habituales de teoría de conjuntos. Por tanto, para cualquier suceso A corresponde un suceso $\overline{A}$ que consiste en todos los puntos muestrales de S que no están en A. Claramente $\overline{A}$ es el suceso «A no sucede». De manera similar el suceso $A \cup B$ se corresponde con la frase «sucede o bien A o bien B», mientras que el suceso $A \cap B$ se corresponde con «suceden tanto A como B». Se dice que dos sucesos son *mutuamente excluyentes* si $A \cap B = \emptyset$.

Por último, una *medida de probabilidad* es una función que asigna un valor numérico Pr[A] a todo suceso A del espacio muestral. Evidentemente, se supone que la probabilidad de un suceso mide en algún sentido la posibilidad relativa de que ese suceso se produzca si se lleva a cabo el experimento aleatorio subyacente. Las bases filosóficas de la noción de probabilidad son controvertidas (¿qué significa precisamente decir que la probabilidad de lluvia mañana es de 0,25?), pero existe un acuerdo en lo tocante a que toda medida sensata de probabilidad debe de satisfacer los tres axiomas siguientes:

1. Para todo suceso A, Pr $[A] \geq 0$
2. Pr $[S] = 1$
3. Si los sucesos A y B son mutuamente excluyentes, esto es, si $A \cap B = \emptyset$, entonces Pr $[A \cup B] =$ Pr $[A]$ + Pr $[B]$

Se sigue por inducción matemática del axioma 3 que

$$\Pr[A_1 \cup A_2 \cup \ldots \cup A_n] = \Pr[A_1] + \Pr[A_2] + \ldots + \Pr[A_n]$$

para toda colección finita $A_1, A_2, \ldots, A_n$ de sucesos mutuamente excluyentes. (Se necesita una forma modificada del axioma 3 si el espacio muestral es infinito, pero no es preciso que nos preocupemos aquí de eso.) Los axiomas dan lugar a un cierto número de consecuencias, entre las cuales se cuentan:

4. Pr $[\overline{A}] = 1 - \Pr[A]$ para todo suceso A ; y
5. Pr $[A \cup B] =$ Pr $[A]$ + *Pr* $[B] - Pr[A \cap B]$ para todo suceso A y B

El procedimiento básico para resolver problemas en probabilidad se puede esbozar a continuación: en primer lugar, se identifica el espacio muestral S; en segundo lugar, se asignan probabilidades a los elementos de S; en tercer lugar, se identifican los sucesos de interés y, finalmente, se calculan las probabilidades deseadas. Por ejemplo, supongamos que se desea conocer la probabilidad de que un generador de números aleatorios produzca un valor que sea primo. Primero tenemos que identificar el espacio muestral. Supongamos que se sabe que el generador puede producir cualquier valor entero entre 0 y 9.999, ambos inclusive. El espacio muestral (esto es, el conjunto de consecuencias posibles) es por tanto { 0, 1, 2,..., 9.999}. A continuación, es preciso asignar probabilidades a los elementos de S. Si el generador produce cada suceso elemental posible con igual probabilidad, se sigue

que Pr [0] = Pr [1] = ... = Pr [9.999] = 1/ 10.000. En tercer lugar, el suceso interesante es «el generador produce un número primo», lo cual se corresponde con el subconjunto A = { 2, 3, 5,..., 9.967, 9.973}. Por último, la probabilidad interesante es Pr [A], que se puede calcular en la forma como $\sum_{e \in A}$ Pr [e], en donde la suma es sobre los sucesos elementales que componen A. Puesto que la probabilidad de todos los sucesos elementales es 1/ 10.000, y hay 1.229 elementos en A, encontramos que Pr [A] = 0,1229.

Hasta el momento hemos supuesto que todo lo que sabemos acerca del resultado de algún experimento aleatorio es que se debe corresponder con algún punto muestral del espacio muestral S. Sin embargo, suele resultar útil calcular la probabilidad de que un suceso A se produzca cuando se sabe que el resultado del experimento está contenido en algún subconjunto B del espacio muestral. Por ejemplo, quizá deseemos calcular la probabilidad de que el generador de números aleatorios del párrafo anterior haya producido un número primo cuando sabemos que ha producido un número impar. El símbolo de esta probabilidad es Pr [A | B], que es lo que se denomina la *probabilidad condicional de A dado B*. Evidentemente esta probabilidad condicional solamente es interesante cuando Pr [B] ≠ 0.

En esencia, la información adicional nos dice que el resultado del experimento pertenece a un nuevo espacio muestral, a saber, B. Puesto que la suma de las probabilidades de los sucesos elementales en el espacio muestral debe de ser 1, incrementamos la escala de los valores originales para cumplir esta condición. Además, solamente estamos interesados ahora en aquella parte de A que está contenida en B, a saber $A \cap B$. Por tanto la probabilidad condicional está dada por:

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}$$

En nuestro ejemplo, A es el suceso «el generador produce un número primo» y B es el suceso «el generador produce un número impar». Sabemos que B contiene 5.000 sucesos elementales, así que $Pr[B] = 0{,}5$. Hay 1.228 primos impares menores que 10.000 (sólo el número primo par 2 se elimina), así que Pr [$A \cap B$] = 0,1228. Por tanto la probabilidad de que el generador haya producido un primo, supuesto que haya producido un número impar, es

$$\Pr[A \mid B] = 0{,}1228 / 0{,}5 = 0{,}2456$$

Se dice que dos sucesos A y B son *independientes* si

$$\Pr[A \cap B] = \Pr[A] \Pr[B]$$

En este caso, suponiendo que $Pr[B] \neq 0$, tenemos que

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]} = \frac{\Pr[A]\Pr[B]}{\Pr[B]} = \Pr[A].$$

y por tanto saber que «el suceso B se ha producido» no modifica la probabilidad de que vaya a producirse el suceso A. De hecho, se puede utilizar esta condición como definición alternativa de independencia.

En el Capítulo 10 examinaremos algoritmos probabilistas para determinar si un entero dado es o no primo. En este contexto resultan útiles las aproximaciones siguientes. Supongamos que el espacio muestral S contiene un gran número n de enteros consecutivos, y que la probabilidad de cada uno de estos resultados es la misma, a saber $1/n$. Por ejemplo, S podría ser el conjunto $\{1, 2,\ldots, n\}$. Sea D_i el suceso «el resultado es divisible por i». Puesto que el número de consecuencias que hay en S y que son divisibles por i es aproximadamente n/i, tenemos que $Pr[D_i] \approx (n/i) \times (1/n) = 1/i$. Además, si p y q son dos primos distintos, entonces los sucesos D_p y D_q son aproximadamente independientes, así que

$$\Pr[D_p \cap D_q] \approx \Pr[D_p] \Pr[D_q] \approx 1/pq$$

Por supuesto esto puede no ser cierto si o bien p o bien q no son primos: claramente los sucesos D_2 y D_4 no son independientes; también cuando S contenga solamente un pequeño número de elementos, las aproximaciones no funcionarán muy bien, según ilustra el problema 1.46. Ahora bien el Teorema de los Números Primos (cuya demostración queda mucho más allá del alcance de este libro) nos dice que el número de primos menores que n es aproximadamente $n / \log n$, así que si S es realmente $\{1, 2,\ldots, n\}$, con igual probabilidad para todos los resultados y A es el suceso «el resultado es primo», entonces tenemos que $\Pr[A] \approx 1/\log n$.

Considérese por ejemplo el problema siguiente: Un generador de números aleatorios produce el valor 12.262.409. Desearíamos saber si este número es primo, pero no disponemos de la potencia de cálculo necesaria para calcular la respuesta de forma determinista. (Por supuesto esto no es real para un ejemplo tan pequeño.) Entonces, ¿qué podemos decir con ayuda de un poco de la teoría de la probabilidad?

Lo primero que hay que observar es que una pregunta como «¿cuál es la probabilidad de que 12.262.409 sea primo?» carece de significado. Aquí no está implicado ningún experimento aleatorio, así que no tenemos un espacio muestral del que hablar, y ni siquiera podemos empezar a asignar probabilidades a los resultados. Por otro lado, una pregunta como «¿cuál es la probabilidad de que nuestro generador de números aleatorios haya producido un número primo?» *es* significativa y se puede responder con bastante facilidad.

Como antes, el primer paso es identificar nuestro espacio muestral, y el segundo es asignar probabilidades a los elementos de S. Supongamos que sabemos que el generador produce todos los valores desde 0 hasta 99.999.999 con igual probabilidad: entonces $S = \{0, 1,\ldots, 99.999.999\}$ y la probabilidad de todos los sucesos elementales de S es $1/10^8$. El Teorema de los Números Primos nos dice que aproximadamente $10^8 / \log 10^8 \approx 5{,}43 \times 10^6$ elementos de S son primos. (El valor exacto es en realidad 5.761.455, pero la aproximación es suficientemente buena para nuestros propósitos.) Por tanto si A es el suceso «nuestro generador produce un número primo» tenemos que $Pr[A] \approx 0{,}0543$.

Ahora consideremos el suceso *Dp* consistente en que «nuestro generador produce un número que es divisible por el primo *p*». Puesto que Pr [Dp] ≈ $1/p$, la probabilidad del suceso complementario «nuestro generador produce un número que *no* es divisible por el primo *p*» es Pr [$\overline{D}$ *p*] ≈1 – $1/p$. Supongamos que probamos el 12.262.409 intentando dividirlo por 2. El intento falla, pero ahora tenemos algo de información adicional, y podemos preguntarnos: «¿cuál es la probabilidad de que nuestro generador produzca un número primo, dado que produce un número que no es divisible por 2?». En forma simbólica:

$$\Pr[A \mid \overline{D}_2] = \Pr[A \cap \overline{D}_2]_{/\,Pr[\,\overline{D}_2]}$$

$$\approx 2 \Pr[A] \approx 0{,}109$$

Aquí *Pr* [$A \cap \overline{D}_2$] es esencialmente lo mismo que *Pr* [*A*] porque todos los primos menos uno son no divisibles por 2. Si a continuación intentamos dividir 12.262.409 por 3, el intento falla una vez más, así que ahora podemos preguntarnos «¿cuál es la probabilidad de que nuestro generador produzca un primo, dado que produce un número que no es divisible ni por 2 ni por 3?». Esta probabilidad es:

$$\Pr[A \mid \overline{D}_2 \cap \overline{D}_3] = \frac{\Pr[A \cap \overline{D}_2 \cap \overline{D}_3]}{\Pr[\overline{D}_2 \cap \overline{D}_3]}$$

$$\approx \Pr[A] \,/\, Pr[\,\overline{D}_2]_{Pr}[\,\overline{D}_3]$$

$$\approx 3 \Pr[A] = 0{,}163$$

Continuando de esta forma, cada fallo sucesivo al intentar dividir 12.262.409 por un nuevo número primo, nos permite hacer una pregunta más precisa, y tener algo más de confianza en que nuestro generador haya producido realmente un número primo. Supongamos, sin embargo, que en alguna fase intentamos dividir 12.262.409 por 3.121. Ahora la división de prueba tiene éxito, así que nuestra próxima pregunta sería: «¿cuál es la probabilidad de que nuestro generador haya producido un número primo, dado que ha producido un número que no es divisible por 2, 3,..., pero si es divisible por 3.121?». La respuesta desde luego es 0: una vez que hemos encontrado un divisor del número producido por el generador, estamos seguros de que no es primo. Simbólicamente, esta respuesta se obtiene porque

$$\Pr[A \cap \overline{D_2} \cap \ldots \cap D_{3.121}] = 0$$

Obsérvese que no podemos comenzar este proceso de cálculo de una nueva probabilidad después de cada división de prueba si no somos capaces de estimar Pr [*A*],

la probabilidad incondicional de que nuestro generador haya producido un número primo. En el ejemplo lo hacíamos utilizando el Teorema de los Números Primos, más nuestro conocimiento de que el generador produce todos los valores desde 0 hasta 99.999.999 con igual probabilidad. Supongamos por otra parte que se nos presenta el número 12.262.409 y se nos dice simplemente que ha sido seleccionado a efectos de este ejemplo. Entonces la pregunta es: «¿cuál es la probabilidad de que un número seleccionado de forma no especificada a efectos de un ejemplo sea primo?». Claramente, es imposible responder a esto. El espacio muestral del experimento aleatorio consistente en seleccionar un número para servir como ejemplo es desconocido, así como también lo son las probabilidades asociadas a los sucesos elementales de este espacio muestral. Por tanto *no* podemos hacer ninguna afirmación significativa acerca de las probabilidades asociadas con un número seleccionado de esta manera.

En muchos experimentos aleatorios nos interesa menos el resultado en sí que algún número que esté asociado con este resultado. En una carrera de caballos, por ejemplo, puede interesarnos menos saber el nombre del caballo vencedor que la cantidad que ganamos o perdemos en la carrera. Esta idea la captura la noción de *variable aleatoria*. Formalmente, una variable aleatoria es una función (y no una variable a pesar de su nombre) que asigna un número real a cada punto muestral de algún espacio muestral S.

Si X es una variable aleatoria definida sobre un espacio muestral S, y x es un número real, definimos el suceso A_x como el subconjunto de S que consta de todos los puntos muestrales a los cuales se les asigna el valor x mediante la variable aleatoria X. De esta manera:

$$A_x = \{s \in S \mid X(s) = x\}$$

La notación $X = x$ es una forma muy cómoda de denotar el suceso A_x. De esta manera, se puede escribir

$$\Pr[X = x] = Pr[Ax] = \sum_{\substack{s \in S \\ X(s)=x}} \Pr(s)$$

Si definimos $p(x)$ como

$$p(x) = Pr[X = x]$$

entonces $p(x)$ es una nueva función asociada a la variable aleatoria X, que se llama la *función de probabilidad de masa* de X. Definimos la *esperanza* $E[X]$ de X (que también se llama *media* o *promedio*) mediante

$$E[X] = \sum_{s \in S} X(s)\, Pr[s] = \sum_{x} xp(x)$$

La esperanza $E[X]$ se denota también frecuentemente como μ_x.

Para reunir todas estas ideas, considérese el ejemplo siguiente. El experimento aleatorio que nos concierne es una carrera de caballos con cinco corredores. La consecuencia del experimento es el nombre del ganador. El espacio muestral S es el conjunto de todos los resultados posibles, así que podríamos tener, por ejemplo:

$$S = \{\text{Ariel, Bombón, Café, Demonio, Ernesto}\}$$

Tenemos que asignar una medida de probabilidad a cada consecuencia posible. Aun cuando la precisión con la cual hacemos esto puede dar lugar a grandes diferencias a nuestra situación financiera, la teoría de la probabilidad no ofrece ayuda acerca de la forma de hacerlo. Supongamos que a la luz de la experiencia asignamos los valores que se muestran en la figura 1. 10.

Resultado	Probabilidad	Ganancias
Ariel	0,10	50
Bombón	0,05	100
Café	0,25	–30
Demonio	0,50	–30
Ernesto	0,10	15

Figura 1.10. Probabilidad de cada consecuencia

(Recuérdese que las probabilidades deben de sumar 1). Hemos hecho un número de apuestas en la carrera, así que, dependiendo del resultado, ganaremos o perderemos algún dinero. La cantidad que ganemos o perdamos también se muestra en la figura 1.10. Esta cantidad es una función del resultado, esto es, una variable aleatoria. Llamemos W a esta variable aleatoria; entonces la tabla muestra que W (Ariel) = 50, W (Bombón) = 100, y así sucesivamente. La variable aleatoria W puede tomar los valores –30, 15, 50 ó 100, y por ejemplo

$$\begin{aligned} p(-30) &= Pr\,[W = -30] \\ &= \sum_{\substack{s \in S \\ W(s) = -30}} \Pr\,[s] \\ &= \Pr\,[\text{Café}] + Pr\,[\text{Demonio}] = 0{,}25 + 0{,}50 = 0{,}75. \end{aligned}$$

De manera similar $p(15) = Pr\,[W = 15] = 0{,}10$, $p(50) = Pr\,[W = 50] = 0{,}10$ y $p(100) = Pr\,[W = 100] = 0{,}05$. Nuestras ganancias esperadas se pueden calcular en la forma

$$\begin{aligned} E[W] &= \sum_x xp(x) \\ &= -30p\,(-30) + 15p\,(15) + 50p\,(50) + 100p\,(100) \\ &= -11 \end{aligned}$$

Una vez que hemos obtenido $E[X]$ podemos calcular una segunda medida útil que se denomina la *varianza* de X, y que se denota como var$[X]$ o σ_x^2. Lo cual se define como

$$Var[X] = E\,[(X - E[\,X\,])^2] = \sum_x p\,(x)\,(x - E\,[X])^2.$$

Dicho en palabras, es el valor esperado del cuadrado de la diferencia entre X y su valor esperado $E[X]$. La desviación estándar de X, que se denota como σ_x, es la raíz cuadrada de la varianza. Para el ejemplo de la carrera de caballos anterior, tenemos:

$$\begin{aligned} \text{var}\,[W] &= \sum_x p(x)(x - E[X])^2 \\ &= p(-30) \times 19^2 + p(15) \times 26^2 + p(50) \times 61^2 + p(100) \times 111^2 \\ &= 1.326{,}5 \end{aligned}$$

y $\sigma_w = \sqrt{(1.326{,}5)} \approx 36{,}42$.

¿Por qué son útiles el valor esperado y la varianza de X? Supongamos que el experimento aleatorio subyacente se pudiera repetir muchas veces. La i-ésima repetición del experimento tendrá algún resultado particular o_i, a la cual la función de X le asigna un valor x_i. Supongamos que repetimos el experimento n veces en total. Si n es grande, casi siempre es razonable esperar que las medias observadas para x_i, a saber $\sum_{i=1}^{n} x_1/n$, estén próximas al valor esperado para X, a saber $E[X]$. Sólo en raras circunstancias es probable que esto no sea verdadero. Por tanto $E[X]$ nos permite predecir el valor medio observado para x_i.

La varianza nos sirve para cuantificar lo buena que probablemente sea esta predicción. Existe una famosa distribución de probabilidad denominada distribución *normal*.

En condiciones muy generales, el *Teorema Central del Límite* sugiere que cuando n es grande, el valor medio observado para x_i tendrá una distribución que es aproximadamente normal con una media $E[X]$ y una varianza var $[X]/n$. Para aprovechar estos resultados lo único que necesitamos es una tabla de la distribución normal. Están disponibles en todas partes. Tales tablas nos dicen, entre otras cosas, que una desviación normal está situada entre más/menos 1,960 desviaciones estándar de su media el 95% de tiempo; 99% del tiempo se encuentra dentro de más/menos 2,576 desviaciones estándar de su media.

Por ejemplo, supongamos que por algún medio mágico la carrera de caballos descrita anteriormente se pudiera efectuar 50 veces en idénticas condiciones.

Supongamos que ganamos w_i en la i-ésima repetición, y sea nuestra ganancia media $\bar{w} = \sum_{i=1}^{50} \bar{w}_i/50$. Entonces podemos esperar que $E[W] = -11$. La varianza de $\vec{w}$ será $\text{Var}[\vec{w}]/50 = 1.326{,}5/50 = 26{,}53$, y su desviación estándar será la raíz cuadrada de esta cantidad, o sea 5,15 mientras que el Teorema Central del Límite nos dice que la distribución de $\bar{w}$ será aproximadamente normal. Por tanto en el 95% de las ocasiones podemos esperar que nuestra ganancia media $\bar{w}$ se encuentre en $-11 - 1{,}960 \times 5{,}15$ y $-11 + 1{,}960 \times 5.15$, eso es entre –21,1 y –0,9. Un cálculo similar muestra que el 99% del tiempo $\vec{w}$ se encontrará entre –24,3 y +2,3; véase el problema 1.47.

Suele ser correcto utilizar el Teorema Central del Límite cuadrado n tiene un valor mayor que aproximadamente 25.

1.8 PROBLEMAS

Problema 1.1. La palabra «álgebra» está también conectada con el matemático al-Khowârizmî, que dio su nombre a los algoritmos. ¿Cuál es la relación?

Problema 1.2. El domingo de Pascua es en principio el primer domingo después de la primera luna llena después del equinoccio de primavera. ¿Es esta regla suficientemente precisa para llamarla un algoritmo? Justifique su respuesta.

Problema 1.3. Mientras se está ejecutando un algoritmo a mano hay que efectuar una elección aleatoria. Para servirse de ella como ayuda, dispone uno de una moneda imparcial, que proporciona los valores *cara* y *cruz* con igual probabilidad, y un dado imparcial que suministra cada uno de los valores del 1 al 6 con igual probabilidad. Se pide seleccionar cada uno de los valores *rojo*, *amarillo* y *azul* con igual probabilidad. Dé al menos tres formas de conseguirlo. Repita el problema con cinco colores en lugar de hacerlo con tres.

Problema 1.4. ¿Es posible que exista un algoritmo para jugar un juego perfecto de billar? Justifique su respuesta.

Problema 1.5. Utilice la multiplicación *à la russe* para multiplicar (a) 63 por 123, y (b) 64 por 123.

Problema 1.6. Busque una calculadora de bolsillo que tenga una precisión mínima de ocho dígitos, esto es, que pueda multiplicar una cifra de cuatro dígitos por otra de cuatro dígitos y obtener la respuesta correcta de ocho dígitos. Se pide multiplicar 31.415.975 por 8.182.818. Mostrar que el algoritmo divide y vencerás de la Sección 1.2 se puede utilizar para solucionar el problema con un pequeño número de cálculos que se pueden hacer en la calculadora, seguido de una pequeña suma con papel y lápiz. Efectúe el cálculo. *Sugerencia:* ¡No lo haga de forma recursiva!

Problema 1.7. Se pide multiplicar dos números dados en números romanos. Por ejemplo, XIX veces XXXIV es DCXLVI. *No* se puede utilizar un método que implique traducir los números a notación arábiga, multiplicarlos y volver a traducirlos después. Diseñe un algoritmo para este problema.

Pista: Busque maneras de traducir hacia adelante y hacia atrás entre la notación romana verdadera y algo parecido que no implique restas. Por ejemplo, XIX podría escribirse como XVIIII en esta notación

«pseudorromana». A continuación, busque formas sencillas de duplicar, partir por dos y sumar cifras en notación pseudorromana. Finalmente, adapte la multiplicación *à la russe* para completar el problema.

Problema 1.8. Igual que en el problema 1.6, suponga que tiene a mano una calculadora de bolsillo que puede multiplicar un número de cuatro cifras por otro de cuatro cifras y obtener la respuesta correcta de ocho cifras. Diseñe un algoritmo para multiplicar dos números grandes basado en el algoritmo clásico, pero utilizando bloques de cuatro cifras de una vez en lugar de sólo una. (Si le parece, piense que está haciendo el cálculo con aritmética en base 10.000.) Por ejemplo, cuando esté multiplicando 1.234.567 por 9.876.543, podría obtener la distribución mostrada en la figura 1.11.

```
                0123  4567
                0987  6543
-------------------------------
      0080      7777  1881
0012  1851      7629
-------------------------------
0012  1932     54061  1881
```

Figura 1.11 Multiplicación en base 10.000

En dicha figura la primera línea del cálculo se obtiene, de derecha a izquierda, en la forma

$$4.567 \times 6.543 = 29.881.881$$

(esto es, un resultado de 1.881 y un arrastre de 2.988), seguido por $0123 \times 6543 + 2988 = 807777$, esto es un resultado de 7777 y un arrastre de 0080). La segunda línea de la multiplicación se obtiene de forma simular, y el resultado final se obtiene sumando las columnas. Toda la aritmética necesaria se puede efectuar con su calculadora.

Utilice el algoritmo para multiplicar 31.415.975 por 8.182.818. Compruebe que su respuesta es la misma que la encontrada en el problema 1.6.

Problema 1.9 La figura 1.12 muestra otro método más para multiplicar dos enteros positivos, que es denominado en algunas ocasiones multiplicación arábiga.

En la figura, tal como antes, 981 se multiplica por 1.234. Para utilizar este método, se dibuja un rectángulo con tantas figuras como cifras haya en el multiplicando (aquí, 3) y con tantas columnas como cifras haya en el multiplicador (aquí, 4). Se escribe el multiplicando encima de las columnas, y el multiplicador hacia abajo en el lado derecho del rectángulo. Se dibujan líneas diagonales según se muestra. A continuación se rellena cada celda del rectángulo con el producto de la cifra que se encuentre en la parte superior de la columna por la cifra que se encuentra en el extremo derecho de la fila. La cifra de las decenas del resultado (que puede ser 0) se pone encima de la línea diagonal,y la cifra de las unidades se ponen por debajo. Por último se suman las diagonales del rectángulo comenzando por la esquina inferior derecha. Aquí la primera diagonal da 4; la segunda da 2 + 0 + 3 = 5; la tercera da 6 + 3 + 4 + 0 + 2 = 15, así que se escribe un 5 y llevamos 1; y así sucesivamente. Ahora el resultado 1.210.554 se puede ir leyendo hacia abajo por el lado izquierdo del rectángulo y también por la fila inferior del rectángulo.

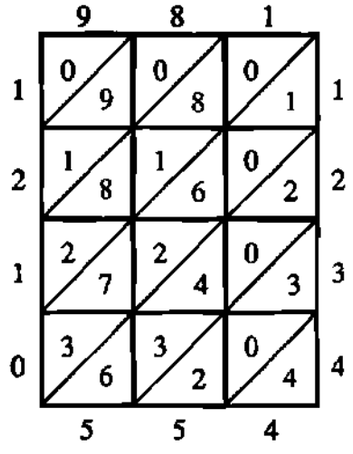


Figura 1.12 Multiplicación empleando un rectángulo

Una vez más, utilice este algoritmo para multiplicar 31.415.975 por 8.182.818, comprobando el resultado con las respuestas de los problemas anteriores.

Problema 1.10. ¿Son iguales los conjuntos $X = [1, 2, 3\}$ e $Y = \{ 2, 1, 3\}$?

Problema 1. 11. ¿Cuáles de los conjuntos siguientes son finitos: $\emptyset$,$\{\emptyset\}$, $\mathbb{N}$ $\{\mathbb{N}\}$? ¿Cuál es la cardinalidad de aquellos conjuntos de los anteriores que sean finitos?

Problema 1.12. ¿Para qué valores de las variables booleanas p, q y r es la fórmula booleana $(p \wedge q) \vee (\neg q \wedge r)$ verdadera?

Problema 1.13. Demostrar que

$\neg(\forall x \in X)\ [P(x)]$ es equivalente a $(\exists x \in X)\ [\neg P(x)]$ y
$\neg(\exists x \in X)\ [P(x)]$ es equivalente a $(x \in X)\ [\neg P(x)]$.

Problema 1.14. Demostrar que

$\neg\ (\overset{\circ}{\forall}\ x \in X)\ [P(x)]$ es equivalente a $(\overset{\circ}{\exists}\ x \in X)\ [\neg P(x)]$ y
$\neg\ (\overset{\circ}{\exists}\ x \in X)\ [P(x)]$ es equivalente a $(\overset{\circ}{\forall}\ x \in X)\ [\neg P(x)]$.

Problema 1.15. Demostrar que

$\log_a (xy) = \log_a x + \log_a y$
$\log_a x^y = y \log_a x$
$\log_a x = \frac{\log_b x}{\log_b a}$
$yx^{\log_b y} = y^{\log_b x}$

Problema 1.16. Demostrar que $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$ para todo número real x.

Problema 1.17. Una demostración alternativa del teorema 1.5.1, consistente en afirmar que hay infinitos números primos, comienza de la forma siguiente. Supongamos para realizar la demostración por contradicción, que el conjunto de los números primos es finito. Sea p el mayor número primo. Considérese $x = p!$ e $y = x + 1$. El problema es completar la demostración desde aquí y derivar a partir de esta demostración un algoritmo *Primomayor* (p) que busque un primo mayor que p. La demostración de terminación para este algoritmo debe ser evidente, así como el hecho consistente en que proporciona un valor mayor que p.

Problema 1.18 Modificar la demostración del teorema 1.5.1 para demostrar que existen infinitos números primos de la forma $4k - 1$, en donde k es un entero.

Sugerencia: Defina x como en la demostración del teorema 1.5.1, pero haga $y = 4x - 1$ en lugar de $y = x + 1$. Aun cuando y en sí puede no ser primo y el menor entero d mayor que 1 que divida a y puede no ser de la forma $4k - 1$, demuestre *por contradicción* que y tiene al menos un divisor primo de la forma requerida.

También es cierto que existen infinitos primos de la forma $4k + 1$, pero esto resulta más difícil de probar. ¿En qué punto se rompe el razonamiento para el caso $4k - 1$ cuando se intenta utilizar la misma idea para demostrar el caso $4k + 1$?

Problema 1.19. Sea n un entero positivo. Dibujar un círculo y marcar n puntos espaciados regularmente alrededor de la circunferencia. Ahora, dibujar una cuerda dentro del círculo entre cada pareja de puntos. En el caso $n = 1$, no hay parejas de puntos y por tanto no se dibuja ninguna cuerda; véase la figura 1.13.

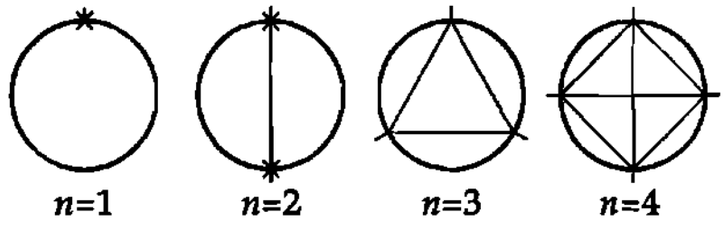


Figura 1.13. Particiones de un círculo

Por último, denotemos mediante $c(n)$ el número de secciones que se construyen así dentro del círculo. Debería encontrarse que $c(1) = 1$, $c(2) = 2$, $c(3) = 4$ y $c(4) = 8$. Por inducción, ¿cuál cree usted que será la fórmula general para $c(n)$? Determine $c(5)$ dibujando y contando. ¿Era correcta la fórmula hallada inductivamente? Vuelva a intentarlo con $c(6)$. ¿Qué pasa si se permite que los puntos se espacíen irregularmente? (opcional y mucho más difícil: determinar la fórmula correcta para $c(n)$, y demostrar que es correcta).

Problema 1.20. ¿Por qué cree que la inducción matemática ha recibido este nombre aun cuando es realmente una técnica deductiva?

Problema 1.21. Demostrar por inducción matemática que la suma de los cubos de los n primeros enteros positivos es igual al cuadrado de la suma de estos enteros.

Problema 1.22. Siguiendo el problema 1.21, demuestre que la suma de los cubos de los n primeros enteros positivos es igual al cuadrado de la suma de estos enteros, pero utilice ahora la proposición 1.7.13 en lugar de la inducción matemática (¡desde luego, la proposición 1.7.13 también se demostró por inducción matemática!)

Problema 1.23. Determinar por inducción todos los valores enteros positivos de n para los cuales $n^3 > 2^n$. Demuestre su afirmación por inducción matemática.

Problema 1.24. El *axioma del mínimo entero* dice que todo conjunto no vacío de enteros positivos contiene un elemento mínimo. (Esta propiedad *no* es general para conjuntos de números reales positivos; considérese por ejemplo el conjunto de todos los enteros que se encuentran estrictamente entre 0 y 1.) Utilizando este axioma, dar una demostración rigurosa por contradicción de que el principio de inducción matemática simple es válido. Más precisamente, considerar cualquier entero a y cualquier propiedad P de los enteros tal que $P(a)$ es cierta, y tal que $P(n)$ es cierta siempre que $P(n-1)$ sea cierta para cualquier entero $n > a$. Supóngase además que *no* es cierto que $P(n)$ sea válido para todo $n \geq a$. Utilizar el axioma del mínimo entero para derivar una contradicción.

Problema 1.25. El problema 1.24 pedía demostrar la validez del principio de inducción matemática a partir del axioma del mínimo entero. De hecho el principio y el axioma son equivalentes: ¡demuestre el axioma del mínimo entero por inducción matemática!

Sugerencia: Como primer paso, demuestre que todo conjunto no vacío de enteros positivos contiene un elemento mínimo por inducción matemática sobre el número de elementos del conjunto. Observe que su demostración sería igualmente válida para cualquier conjunto finito de números reales, lo cual muestra claramente que no es aplicable directamente a los conjuntos infinitos. Para generalizar el resultado a un conjunto infinito de números enteros, considere cualquiera de estos conjuntos X. Sea m cualquier elemento de X (no se necesita un axioma para esto: todo conjunto infinito contiene al menos un elemento por definición). Sea Y el conjunto de elementos de X tales que no son mayores que m. Mostrar que Y es un conjunto de enteros positivos no vacío y finito, y consiguientemente la demostración por inducción matemática será aplicable para concluir que Y contiene un elemento más pequeño, digamos n. Finalice la prueba argumentando que n es también el menor elemento de X.

Problema 1.26. Dar una demostración rigurosa de que el principio *generalizado* de inducción matemática es válido.

Sugerencia: Demuéstrelo por inducción matemática sencilla.

Problema 1.27. Recuerde que la *sucesión de Fibonacci* está definida en la forma

$$\begin{cases} f_0 = 0; f_1 = 1; & \text{y} \\ f_n = f_{n-1} + f_{n-2} & \text{para } n \geq 2 \end{cases}$$

Demostrar por inducción matemática generalizada que

$$f_n = \frac{1}{\sqrt{5}} [\phi^n - (-\phi)^{-n}]$$

en donde

$$\phi = \frac{1+\sqrt{5}}{2}$$

que es la *razón áurea*. (Esto se conoce con el nombre de fórmula de Moivre.)

Problema 1.28. Siguiendo el problema 1.27, demostrar por inducción matemática que $f_n > \left(\frac{3}{2}\right)^n$ para todo entero n suficientemente grande. ¿Cómo de grande tiene que ser n? *No* utilice la fórmula de Moivre.

Problema 1.29. Siguiendo el problema 1.27, demostrar que

$$\sum_{i=0}^{k} \binom{k}{i} f_{n+i} = f_{n+2k}$$

Problema 1.30. Siguiendo el problema 1.27, demostrar por inducción matemática generalizada que $f_{2n+1} = f_n^2 + f_{n+1}^2$ para todos los enteros $n \geq 0$.

Sugerencia: Demostrar también el teorema más fuerte consistente en que

$$f_{2n} = 2 f_n f_{n+1} - f_n^2$$

Problema 1.31. Considérense las constantes arbitrarias reales y positivas a y b. Defínase por recurrencia $t: \mathbb{N}^+ \rightarrow \mathbb{R}^+$

$$\begin{cases} t(1) = a & \cdots \quad \text{y} \\ t(n) = bn + nt(n-1) & \text{para } n \geq 2 \end{cases}$$

Esta función es tan parecida a la recurrencia $n! = n \times (n-1)!$ que caracteriza al factorial, que resulta natural conjeturar la existencia de dos constantes reales y positivas u y v tales que $un! \leq t(n) \leq vn!$ para cada entero n suficientemente grande. Demostrar esta conjetura por inducción constructiva.

Sugerencia: Probar la sentencia más fuerte consistente en que existen tres constantes reales positivas u, v y w tales que $un! \leq t(n) \leq vn! - wn$ para cada entero n suficientemente grande.

Nota: Este problema muestra que el tiempo requerido para utilización directa de la definición recursiva para calcular el determinante de una matriz $n \times n$ es proporcional a $n!$, lo cual es mucho peor que si el tiempo fuera meramente exponencial. Por supuesto, el determinante se puede calcular de forma más eficiente por eliminación de Gauss-Jordan. Véase la Sección 2.7.1 para más detalles.

Problema 1.32. Vuelva a hacer el problema 1.31, pero defina ahora $t(n) = bn^k + nt(n-1)$ para $n \geq 2$, en donde k es una constante entera positiva arbitraria.

Problema 1.33. Considere una constante real positiva arbitraria d y una función $t: \mathbb{N} \rightarrow \mathbb{R}^+$ tal que

$$t(n) \leq dn + \frac{2}{n} \sum_{k=0}^{n-1} t(k)$$

para cada entero positivo n. Utilizando la inducción constructiva, demostrar la existencia de una constante c real y positiva tal que $t(n) \leq cn \log n$ para cada entero n suficientemente grande.

Problema 1.34: Dar dos funciones $f(n)$ y $g(n)$ tales que ni $f(n)$ ni $g(n)$ tiendan a un lí-

mite cuando n tiende a infinito, pero tales que, ambos, $f(n) + g(n)$ y $f(n) / g(n)$ tiendan realmente a un límite.

Problema 1.35. Determinar el comportamiento cuando n tiende a infinito de $f(n) = n^{-r} x^n$, donde r es cualquier entero positivo.

Problema 1.36. Utilizar la regla de l'Hôpital para hallar el límite cuando n tiende a infinito de $(\log \log n)^a / \log n$, en donde $a > 0$ es una constante positiva arbitraria.

Problema 1.37. Demostrar la proposición 1.7.8.

Problema 1.38. Dar una demostración sencilla, sin utilizar integrales, de que la serie armónica diverge.

Problema 1.39. Utilizar una técnica similar a la ilustrada en la figura 1.8 para mostrar que para $r > 1$ la suma es

$$Sn = 1 + \frac{1}{2^r} + \frac{1}{3^r} + \ldots + \frac{1}{n^r}$$

converge a un límite menor que $r/(r-1)$.

Problema 1.40. (Serie alternante) Sea $f(n)$ una función positiva y estrictamente decreciente de n que tiende a 0 cuando n tiende a infinito. Mostrar que la serie $f(1) - f(2) + f(3) - f(4) + \ldots$, es convergente, y que su suma se encuentra entre $f(1) - f(2)$. Por ejemplo:

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots = \log 2 \approx 0{,}69314\ldots$$

Mostrar, más aún, que el error que se comete si se aproxima la suma de toda la serie por la suma de los n primeros términos es menor que $f(n+1)$.

Problema 1.41. Mostrar que $\binom{n}{r} \leq 2^{n-1}$ para $n > 0$ y $0 \leq r \leq n$.

Problema 1.42. Probar que

$$\binom{2n}{n} \geq 4^n / (2n+1).$$

Problema 1.43. Mostrar que $\sum_{r=1}^{n} r\binom{n}{r} = n2^{n-1}$ para $n > 0$.

Sugerencia: Derivar los dos lados del teorema 1.7.20.

$$\frac{1}{(1+x)^2} = 1 - 2x + 3x^2 - 4x^3 + 5x^4 + \ldots \text{ para } -1 < x < 1$$

Sugerencia: Utilizar la proposición 1.7.12.

Problema 1.45. Mostrar que dos sucesos mutuamente excluyentes *no* son independientes salvo en el caso trivial consistente en que al menos uno de ellos tenga la probabilidad cero.

Problema 1.46. Considérese un experimento aleatorio cuyo espacio muestral S es {1, 2, 3, 4, 5}. Sean A y B los sucesos «el resultado es divisible por 2» y «el resultado es divisible por 3», respectivamente. ¿Cuánto valen $Pr[A]$, $Pr[B]$ y $Pr[A \cap B]$? ¿Son los sucesos A y B independientes?

Problema 1.47. Mostrar que para la carrera de caballos de la Sección 1.7.4, el 99% del tiempo nuestras ganancias esperadas $\overline{w}$ promediadas a lo largo de 50 carreras se encuentran entre –24,3 y +2,3.

1.9 REFERENCIAS Y TEXTOS MÁS AVANZADOS

Distinguiremos tres clases de libros que se ocupan del diseño y análisis de algoritmos. Los libros *específicos* abarcan algoritmos que son útiles en un área de aplicación particular: ordenación y búsqueda, teoría de grafos, geometría computacional, etc, etc. Los libros *generales* abarcan varias áreas de aplicación: proporcionan algoritmos útiles para cada área. Por último, los terceros, de *algoritmia*, se concentran en las técnicas de diseño y análisis de algoritmos: ilustran cada técnica mediante ejemplos de algoritmos tomados de distintas áreas de aplicación. La diferencia entre estos dos tipos de libros es necesariamente difusa en algunas ocasiones. Harel (1987) adopta un punto de vista más amplio de la Algoritmia, y la considera nada menos que «el espíritu de la computación».

La colección de algoritmos más ambiciosa que se haya intentado nunca se debe sin duda a Knuth (1968, 1969, 1973), que originalmente tenía la intención de constar de siete volúmenes. Hay varios capítulos del *Handbook of Theoretical Computer Science*, editado por van Leeuwen (1990), que son de gran interés para el estudio de algoritmos. Hay muchos otros libros generales que merece la pena mencionar: por orden cronológico Aho, Hopcroft y Ullman (1974), Baase (1978), Dromey (1982), Sedgewick (1983), Gonnet y Baeza-Yates (1984), Melhorn (1984a, 1984b, 1984c), Manber (1989), Cormen, Leiserson y Rivest (1990), Kingston (1990), Lewis y Danenberg (1991), Kozen (1992) y Nievergelt e Hinrichs (1993).

Haremos alusión a libros concretos en los capítulos siguientes siempre que sean relevantes para nuestra discusión; podríamos mencionar, sin embargo, a Nilsson (1971), Brigham (1974), Borodin y Munro (1975), Christofides (1975), Lawler (1976), Reingold, Nievergelt y Deo (1977), Gondran y Minoux (1979), Even (1980), Papadimitriou y Steiglitz (1982), Tarjan (1983), Akl (1989), Lakshmivarahan y Dhall (1990), Ja´Ja´(1992) y Leighton (1992).

Además de nuestros propios libros —éste y Brassard y Bratley (1988)— y los de Harel, conocemos tres trabajos más acerca de la Algoritmia: Horowitz y Sahni (1978), Stinson (1985) y Moret y Shapiro (1991). Para una descripción más popular de los algoritmos, véase Knuth (1977) y Lewis y Papadimitriou (1978).

La multiplicación *à la russe* se describe en Warusfel (1961), un notable librito francés de matemáticas populares, pero la idea básica era conocida a los antiguos egipcios, quizá ya en el año 3500 a. C.; véase Ahmes (1700 a.C.) y Kline (1972). La multiplicación utilizando «divide y vencerás» ha sido atribuida a Karatsuba y Ofman (1962), mientras que la multiplicación arábiga ha sido descrita en Eves (1983). Para una discusión de las demostraciones matemáticas constructivas y no constructivas tales como las que dimos para el teorema 1.5.2, consúltese Bishop (1972). El principio del descubrimiento matemático ha sido descrito por Pólya (1945, 1954). El primer contraejemplo de la conjetura de Euler acerca de que no es posible que la suma de tres potencias de números enteros positivos sea una cuarta potencia ha sido dado por Elkies (1988).

Aun cuando no utilizamos ningún lenguaje específico de programación en el presente texto, sugerimos que los lectores que no estén familiarizados con Pascal harían bien en examinar algunos de los numerosos libros existentes acerca de este lenguaje, tal como el de Jensen y Wirth (1985) o el de Lecarme y Nebut (1985).

Rosen (1991) es una introducción extensa y simple a temas tales como el cálculo proposicional, conjuntos, la probabilidad, el razonamiento matemático y los grafos.

Leonardo Pisano (*c.* 1170-*c.* 1240), o Leonardo Fibonacci, fue el gran matemático occidental de la Edad Media. Hay una breve descripción de su vida y un segmento del *Liberabaci* en Calinger (1982). La demostración de que

$$\lim_{n\to\infty}\left(1+\frac{1}{2^2}+\frac{1}{3^2}+\ldots\frac{1}{n^2}\right)=\frac{\pi^2}{6}$$

se debe a Euler; véase Eves (1983) o Scharlau y Opolka (1985). Estas referencias contienen también la historia del teorema binomial.

Capítulo 2

Algoritmia* elemental

2.1 INTRODUCCIÓN

En este capítulo comenzaremos nuestro estudio detallado de los algoritmos. Primero definiremos algunos términos: veremos que un *problema*, tal como multiplicar dos enteros positivos, va a tener normalmente muchos, normalmente infinitos, *ejemplares*[2], tales como multiplicar los enteros concretos 981 y 1.234. Los algoritmos deben funcionar correctamente en *todos* los casos del problema que afirman resolver.

A continuación explicaremos lo que queremos decir con la *eficiencia* de un algoritmo, y discutiremos distintas formas de seleccionar el algoritmo más eficiente para resolver un problema cuando están disponibles varias técnicas que compiten entre sí. Veremos que resulta crucial cómo cambia la eficiencia del algoritmo a medida que los casos del problema se vuelven mayores y por tanto (normalmente) más difíciles de resolver. También distinguiremos entre la eficiencia media de un algoritmo cuando se utiliza en muchos casos de un problema y su eficiencia en el peor caso posible. La estimación pesimista, del peor caso posible, suele ser adecuada cuando tenemos que estar seguros de resolver un problema en una cantidad limitada de tiempo, por ejemplo.

Una vez se ha definido lo que queremos decir por eficiencia, podemos empezar a investigar los métodos utilizados para analizar algoritmos. Nuestra línea de ataque consistirá en contar el número de operaciones elementales, tales como sumas y multiplicaciones, que efectúa el algoritmo. Sin embargo, veremos que incluso estas operaciones tan corrientes no son totalmente sencillas: tanto la adición como la multiplicación se vuelven más lentas a medida que aumenta el tamaño de sus operandos. También intentaremos dar alguna noción de la diferencia práctica de un algoritmo bueno y uno malo en términos de tiempo de computación.

---

* Las notas a pie de página se ha preferido agruparlas todas al final del libro, en un apartado llamado *Notas finales*.

Un tema que *no* vamos a abarcar, en este capítulo ni en ningún otro lugar, es la forma de demostrar rigurosamente que los programas que utilizamos para representar los algoritmos son correctos. Esta aproximación requiere una definición formal de la semántica de los lenguajes de programación que va mucho más allá de lo que consideramos necesario: un tratamiento adecuado de este tema merecería un libro en sí. Para nuestros propósitos nos conformaremos con fiarnos de demostraciones informales que utilizan argumentos de sentido común.

El capítulo concluye con un cierto número de ejemplos de algoritmos procedentes de diferentes áreas, algunos buenos y otros malos, para demostrar la forma en que los principios expresados se aplican en la práctica.

2.2 PROBLEMAS Y EJEMPLARES

En la Sección 1.1 hemos presentado varias formas distintas de multiplicar dos enteros positivos, tomando como ejemplo la multiplicación de 981 por 1234. Sin embargo los algoritmos esbozados aquí no se limitan a proporcionar una forma de multiplicar estos dos números en concreto. De hecho, ofrecen una solución general al *problema* consistente en multiplicar dos enteros positivos. Diremos que (981, 1234) es un ejemplar o un caso de este problema. La multiplicación de 789 por 9742 que se puede expresar como (789, 9742), es otro ejemplar del mismo problema. Sin embargo, la multiplicación de –12 por 83.7 no lo es, por dos razones: –12 no es positivo, y 83.7 no es entero. Por supuesto (–12, 83.7) es un ejemplar de otro problema de multiplicación más general. La mayoría de los problemas interesantes tienen una colección infinita de ejemplares. Sin embargo, hay excepciones. Hablando propiamente, el problema de jugar un juego perfecto de ajedrez solamente tiene un ejemplar, puesto que se da una única posición inicial. Además solamente existe un número finito de subcasos (las posiciones válidas intermedias). Sin embargo esto no significa que el problema carezca de interés algorítmico.

Un algoritmo debe funcionar correctamente en todos los ejemplares o casos del problema que manifiesta resolver. Para mostrar que un algoritmo es incorrecto solamente es necesario encontrar un ejemplar del problema para el cual no sea capaz de encontrar una respuesta correcta. Del mismo modo que es posible demostrar que un teorema no es válido encontrando un único contraejemplo, un algoritmo puede rechazarse tomando como base un único resultado incorrecto. Por otra parte, del mismo modo que puede resultar difícil demostrar un teorema, también es difícil normalmente demostrar la corrección de un algoritmo. (Sin embargo, véase la Sección 1.6.3 para un caso sencillo.) Para hacer que esto sea posible, cuando especificamos un problema, es importante definir su *dominio de definición*, esto es, el conjunto de casos que deben considerarse. Los algoritmos de multiplicación que se dan en el Capítulo 1 no funcionarán para algoritmos negativos o fraccionarios, o por lo menos no sin ciertas modificaciones. Sin embargo, esto no significa que los algoritmos no sean válidos: en los casos de multiplicación que implican números negativos o fracciones no se encuentran en el dominio de definición que nosotros seleccionamos al definir el problema.

Todo dispositivo de cálculo real tiene un límite que afecta al tamaño de los casos que puede manejar, bien como consecuencia de que los números implicados sean demasiado grandes, o porque nos quedemos sin espacio. Sin embargo, esta limitación no debe ser achacada al algoritmo que decidamos utilizar. Distintas máquinas tienen distintos límites, e incluso distintos programas que implementen un mismo algoritmo en la misma máquina pueden imponer diferentes limitaciones. En este libro casi siempre nos contentaremos con demostrar que nuestros algoritmos son correctos en abstracto, ignorando las limitaciones prácticas de cualquier programa concreto para implementarlos.

2.3 LA EFICIENCIA DE LOS ALGORITMOS

Cuando tenemos que resolver un problema, es posible que estén disponibles varios algoritmos adecuados. Evidentemente, desearíamos seleccionar el mejor. Esto plantea la pregunta de cómo decidir entre varios algoritmos cuál es preferible. Si solamente tenemos que resolver uno o dos casos pequeños de un problema más bien sencillo, quizá no nos importe demasiado qué algoritmo utilizaremos: en este caso podríamos decidirnos a seleccionar sencillamente el que sea más fácil de programar, o uno para el cual ya exista un programa, sin preocuparnos por sus propiedades teóricas. Sin embargo, si tenemos que resolver muchos casos, o si el problema es difícil, quizá tengamos que seleccionar de forma más cuidadosa.

El enfoque *empírico* (o *a posteriori*) para seleccionar un algoritmo consiste en programar las técnicas competidoras e ir probándolas en distintos casos con ayuda de una computadora. El enfoque *teórico* (o *a priori*) que es el que nosotros propugnamos en este libro, consiste en determinar matemáticamente la cantidad de recursos necesarios para cada uno de los algoritmos *como función del tamaño de los casos considerados*. Los recursos que más nos interesan son el tiempo de computación y el espacio de almacenamiento, siendo el primero normalmente el más importante. A lo largo del libro, por lo tanto, compararemos normalmente los algoritmos tomando como base sus tiempos de ejecución, y cuando hablemos de la *eficiencia* de un algoritmo, querremos decir simplemente lo rápido que se ejecuta. Sólo en algunas ocasiones nos interesarán también los requisitos de almacenamiento de un algoritmo, o su necesidad de otros recursos. (Ejemplos de otros recursos son el número de procesadores que se necesitan por parte de un algoritmo paralelo, y también algunas combinaciones artificiosas pero significativas: quizá nos interese minimizar el producto del espacio de almacenamiento utilizado por el tiempo durante el cual está siendo ocupado, si es que es así la forma en que se calculan nuestras facturas.)

El *tamaño* de un ejemplar se corresponde formalmente con el número de *bits* que se necesitan para representar el ejemplar en una computadora, utilizando algún esquema de codificación precisamente definido y razonablemente compacto. Sin embargo, para hacer más claros nuestros análisis, lo normal será que seamos menos formales, y utilizaremos la palabra «tamaño» para indicar cualquier entero que mida de alguna forma el número de componentes de un ejemplar. Por ejem-

plo, cuando estamos hablando acerca de ordenaciones, mediremos normalmente el tamaño de un ejemplar por el número de ítems que hay que ordenar ignorando el hecho consistente en que cada uno de estos ítems ocuparía más de un *bit* para representarlo en una computadora. De manera similar, cuando hablemos acerca de grafos, mediremos normalmente el tamaño de un ejemplar por el número de nodos o de aristas (o de ambos) implicados. Apartándonos un poco de esta regla general, sin embargo, cuando hablemos acerca de problemas que impliquen enteros, daremos algunas veces la eficiencia de nuestros algoritmos en términos del *valor* del ejemplar que estemos considerando, en lugar de considerar su tamaño (que sería el número de *bits* necesarios para representar en binario este valor).

La ventaja de la aproximación teórica es que no depende ni de la computadora que se esté utilizando, ni del lenguaje de programación, ni siquiera de las habilidades del programador. Se ahorra tanto el tiempo que se habría invertido innecesariamente para programar un algoritmo ineficiente, como el tiempo de máquina que se habría desperdiciado comprobándolo. Lo que es más significativo, se nos permite estudiar la eficiencia del algoritmo cuando se utiliza en casos de todos los tamaños. Esto no suele suceder con la aproximación empírica, en la cual las consideraciones prácticas podrían obligarnos a comprobar los algoritmos sólo en un pequeño número de ejemplares arbitrariamente seleccionados y de tamaño moderado. Dado que suele suceder que los algoritmos recién descubiertos empiezan a comportarse mejor que sus predecesores sólo cuando ambos se utilizan en ejemplares grandes, este último punto resulta especialmente importante.

También resulta posible analizar los algoritmos utilizando un enfoque *híbrido*, en el cual la forma de la función que describe la eficiencia del algoritmo se determina teóricamente, y entonces se determinan empíricamente aquellos parámetros numéricos que sean específicos para un cierto programa y para una cierta máquina, lo cual suele hacerse mediante algún tipo de regresión. Empleando este enfoque se puede predecir el tiempo que necesitará una cierta implementación para resolver un ejemplar mucho mayor que los que se hayan empleado en las pruebas. Sin embargo, hay que tener cuidado cuando se hacen estas extrapolaciones basándose solamente en un pequeño número de comprobaciones empíricas y anulando toda consideración teórica. Las predicciones hechas sin apoyo teórico tienen grandes probabilidades de ser imprecisas, si es que no resultan completamente incorrectas.

Si deseamos medir la cantidad de espacio que utiliza un algoritmo en función del tamaño de los ejemplares, está a nuestra disposición una unidad natural, a saber, el *bit*. Independientemente de la máquina que se esté utilizando, la noción de un *bit* de almacenamiento está bien definida. Si, por otra parte, tal como suele suceder, deseamos medir la eficiencia de un algoritmo, en términos del tiempo que se necesita para llegar a una respuesta, entonces no existe una opción tan evidente. Está claro que no se puede pensar en expresar esta eficiencia, digamos, en segundos, puesto que no se dispone de una computadora estándar a la cual se pudieran referir todas las medidas.

Una respuesta a este problema es la que viene dada por el *principio de invariancia*[3], que afirma que dos implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia en más de alguna constante multiplicativa. Si esta constante fuera, por ejemplo, cinco, entonces sabemos que si la primera implementación requiere un segundo para resolver casos de un cierto tamaño, entonces la segunda implementación (quizá en una máquina distinta, o escrita en un lenguaje de programación distinto) no requerirá más de 5 segundos para resolver los mismos casos. Para ser más exactos, si dos implementaciones del mismo algoritmo necesitan $t_1(n)$ y $t_2(n)$ segundos, respectivamente, para resolver un caso de tamaño n, entonces siempre existen constantes positivas c y d tales que $t_1(n) \leq ct_2(n)$ y $t_2(n) \leq dt_1(n)$ siempre que n sea suficientemente grande. En otras palabras, el tiempo de ejecución de cualquiera de las implementaciones está acotado por un múltiplo constante del tiempo de ejecución de la otra; la decisión de a qué implementación llamaremos primera, y a cuál llamaremos segunda, es irrelevante. La condición de que n sea suficientemente grande no es realmente necesaria: véase la «regla del umbral» en la Sección 3.2. Sin embargo, al incluirla suele ser posible hallar constantes más pequeñas c y d que las que se hallarían en caso contrario. Esto resulta útil si estamos intentando calcular buenas cotas acerca del tiempo de ejecución de una implementación cuando conocemos el tiempo de ejecución de la otra.

Este principio no es algo que se pueda demostrar: simplemente establece un hecho que puede ser confirmado por observación. Además tiene un amplio rango de aplicación. El principio sigue siendo cierto sea cual fuere la computadora utilizada para implementar el algoritmo (siempre y cuando sea de diseño convencional), independientemente del lenguaje de programación y del compilador empleado, independientemente incluso de la habilidad del programador (¡siempre y cuando no se modifique el algoritmo!). De esta manera un cambio de máquina puede permitirnos resolver un problema 10 veces más deprisa ó 100 veces más deprisa, proporcionando un aumento de velocidad describible mediante un factor constante. Un cambio de algoritmo, por otra parte —y sólo un cambio de algoritmo— puede darnos un incremento que se vuelva cada vez más pronunciado a medida que crezca el tamaño de los casos.

Volviendo a la cuestión de la unidad que se debe utilizar para expresar la eficiencia teórica de un algoritmo, el principio de invariancia nos permite decidir que no va a existir tal unidad. En su lugar, expresaremos solamente el tiempo requerido por el algoritmo salvo una constante multiplicativa. Diremos que un algoritmo para algún problema requiere un tiempo del orden de $t(n)$ para una función dada t, si existe una constante positiva c y una implementación del algoritmo capaz de resolver todos los casos de tamaño n en un tiempo que no sea superior a $ct(n)$ segundos. (Para los problemas numéricos, tal como se comentó anteriormente, n puede en algunas ocasiones ser el valor más que el tamaño del caso.)

El uso de segundos en esta definición es evidentemente arbitrario: sólo se necesita modificar la constante para acotar el tiempo por $at(n)$ años o por $bt(n)$ microsegundos. Por el principio de invariancia, si cualquier implementación del al-

goritmo tiene la propiedad requerida, entonces también la tienen todas las demás, aunque la constante multiplicativa pueda cambiar de una implementación a otra. En el capítulo siguiente daremos un tratamiento riguroso de este importante concepto conocido como *notación asintótica*. Quedará claro a partir de la definición formal por qué decimos «en el orden de» en lugar de emplear el más habitual «del orden de».

Hay ciertos órdenes que se producen con tanta frecuencia que merece la pena darles un nombre. Por ejemplo, supongamos que el tiempo necesario para que un algoritmo resuelva un caso del tamaño n nunca es más que cn segundos, en donde c es alguna constante adecuada. Diremos entonces que el algoritmo requiere un tiempo en el orden de n, o más simplemente que requiere un tiempo *lineal*. En este caso también hablamos de un *algoritmo lineal*. Si un algoritmo nunca necesita más de cn^2 segundos para resolver un caso de tamaño n, entonces diremos que requiere un tiempo en el orden de n^2, o bien que requiere un tiempo *cuadrático*, y le llamaremos *algoritmo cuadrático*. De manera similar un algoritmo es *cúbico*, *polinómico* o *exponencial* si requiere un tiempo en el orden de n^3, n^k o c^n, respectivamente, en donde k y c son constantes adecuadas. La Sección 2.6 ilustra las importantes diferencias entre estos órdenes de magnitud.

No caiga en la trampa de olvidar por completo las *constantes ocultas*, nombre éste que suele asignarse a las constantes multiplicativas que se utilizan en estas definiciones. Normalmente se ignoran los valores exactos de estas constantes, y se supone que todos ellos son aproximadamente del mismo orden en magnitud. Esto nos permite decir, por ejemplo, que un algoritmo lineal es más rápido que un algoritmo cuadrático sin preocuparnos acerca de si nuestra afirmación es verdadera en todos los casos. Sin embargo, en algunos casos hay que ser más cuidadoso.

Considérense por ejemplo dos algoritmos cuyas implementaciones en una cierta máquina requieren n^2 días y n^3 segundos para resolver un caso del tamaño n. ¡Solamente en casos que requieran más de 20 millones de años para resolverlos, el algoritmo cuadrático será más rápido que el algoritmo cúbico! (véase el problema 2.7). Desde un punto de vista teórico, el primero es *asintóticamente* mejor que el segundo; esto es, su rendimiento es mejor para todos los casos suficientemente grandes. Sin embargo, desde un punto de vista práctico, preferiremos ciertamente el algoritmo cúbico. Aunque el algoritmo cuadrático pueda ser asintóticamente mejor, su constante oculta es tan grande que lo hace imposible de utilizar para casos de tamaño normal.

2.4 ANÁLISIS DE «CASO MEDIO» Y «CASO PEOR»

El tiempo que requiere un algoritmo, o el espacio de almacenamiento que consume, pueden variar considerablemente entre dos ejemplares distintos del mismo tamaño. Para ilustrar esto, considérense dos algoritmos elementales de ordenación, la ordenación por *inserción* y la ordenación por *selección*:

```
procedimiento insertar (T[1...n ])
   para i ← 2 hasta n hacer
      x ← T[i]; j ← i − 1
      mientras j > 0 y x < T[j] hacer T[j+ 1] ← T[j]
                                      j ← j − 1
      T[j + 1] ← x

procedimiento seleccionar (T[1... n ])
   para i ← 1 hasta n − 1 hacer
      minj ← i; minx ← T[i]
      para j ← i + 1 hasta n hacer
        si T[j] < minx entonces minj ← j
                                minx ← T[j]
      T[minj] ← T[i]
      T[i] ← minx
```

Simule el funcionamiento de estos dos algoritmos en unas pocas matrices para asegurarse de que entiende la forma en que funcionan. El bucle principal de la ordenación por inserción va examinando sucesivamente todos los elementos de la matriz desde el segundo hasta el n-ésimo, e inserta cada uno en el lugar adecuado entre sus predecesores dentro de la matriz[4]. La ordenación por selección funciona seleccionando el menor elemento de la matriz y llevándolo al principio; a continuación selecciona el siguiente menor y lo pone en la segunda posición de la matriz, y así sucesivamente.

Sean U y V dos matrices de n elementos tales que U ya está ordenada por orden ascendente, mientras que V está ordenada por orden descendente. El problema 2.9 muestra que estos dos algoritmos requieren más tiempo en V que en U. De hecho, la matriz V representa el peor caso posible para estos dos algoritmos: ninguna matriz de n elementos requiere más trabajo. Sin embargo, el tiempo requerido por el algoritmo de ordenación por selección no es muy sensible al orden original de la matriz que hay que ordenar: la comprobación «**si** $T[j] < minx$» se ejecuta exactamente el mismo número de veces en todos los casos. La variación de tiempos de ejecución solamente se debe al número de veces que se ejecutan las asignaciones que aparecen en la parte **entonces** de esta comprobación. Cuando programamos este algoritmo y lo comprobamos en una máquina, encontramos que el tiempo necesario para ordenar un número de elementos no variaba en más de un 15% sea cual fuese el orden de los elementos que había que ordenar. Tal como se mostrará en la Sección 4.4, el tiempo requerido por *seleccionar* T es cuadrático, independientemente del orden inicial de los elementos.

La situación es distinta si se comparan los tiempos requeridos tomados por la ordenación por inserción del algoritmo en estas dos matrices. Dado que la condi-

ción que controla el bucle **mientras** siempre es falsa en el momento inicial, *insertar* (*U*) es muy rápido, y requiere un tiempo lineal. Por otra parte *insertar* (*V*) requiere un tiempo cuadrático, porque el bucle **mientras** se ejecuta $i - 1$ veces para todo el valor de i; véase una vez más la Sección 4.4. La variación de tiempo existente en estos dos casos es por tanto considerable. Además, esta variación se incrementa con el número de elementos que haya que ordenar. Cuando implementamos el algoritmo de ordenación por inserción, encontramos que requería menos de un quinto de segundo para ordenar una matriz de 5.000 elementos que ya estaba en orden ascendente, mientras que necesitó tres minutos y medio —esto es, mil veces más— para ordenar una matriz del mismo número de elementos que esta vez se encontraba inicialmente en orden descendente.

Si pueden producirse variaciones tan grandes, ¿cómo se puede hablar acerca del tiempo requerido por un algoritmo en términos únicamente del tamaño del caso que haya que resolver? Normalmente, consideraremos el *caso peor* del algoritmo, esto es, para cada tamaño de caso solamente consideraremos aquellos en los cuales el algoritmo requiera más tiempo. Ésta es la razón por la cual decíamos en la sección precedente que un algoritmo que se ejecuta en un tiempo en el orden de $t(n)$, debe de ser capaz de resolver *todos* los casos de tamaños n en un tiempo no superior a $ct(n)$ segundos, para una constante adecuada n, que dependerá de la implementación: implícitamente teníamos en mente el caso peor.

El análisis en «el caso peor» es adecuado para algoritmos cuyos tiempos de respuesta sean críticos. Por ejemplo, si se trata de controlar una planta nuclear, es crucial saber el límite superior del tiempo de respuesta del sistema, independientemente del ejemplar concreto que haya que resolver. Si, por otra parte, es preciso utilizar muchas veces un algoritmo en muchos casos distintos, quizá sea más importante conocer el tiempo medio de ejecución para ejemplares de tamaño n. Ya vimos que el tiempo requerido por el algoritmo de ordenación por inserción oscilaba entre el orden n y el orden n^2. Si podemos calcular el tiempo medio requerido por el algoritmo en los $n!$ modos distintos de ordenar elementos distintos inicialmente, tendremos una idea del tiempo probable que será necesario para ordenar una matriz que inicialmente se encuentre en orden aleatorio. Veremos en la Sección 4.5 que si las $n!$ permutaciones iniciales son igualmente probables, entonces este tiempo medio también está en el orden de n^2. La ordenación por inserción requiere por tanto un tiempo cuadrático en el caso medio como en el caso peor, aun cuando para algunos casos pueda ser mucho más rápida. En la Sección 7.4.2 veremos otro algoritmo de ordenación que también tiene un tiempo cuadrático en el caso peor, pero que requiere solamente un tiempo en el orden de $n \log n$ por término medio. Aun cuando este algoritmo tiene un caso peor —el rendimiento cuadrático es lento para un algoritmo de ordenación— se trata posiblemente del algoritmo más rápido que se conoce en el caso medio para métodos de clasificación *in situ*, esto es, que no requieren espacio adicional de almacenamiento.

Suele ser más difícil analizar el comportamiento medio de un algoritmo que analizar su comportamiento en el caso peor. Además, semejante análisis de comportamiento medio podría ser confuso si de hecho los casos que hubiera que resolver no se seleccionaran aleatoriamente cuando el algoritmo se utiliza en la práctica. Por ejemplo, hemos afirmado anteriormente que la ordenación por inserción requiere un tiempo cuadrático por término medio cuando todas los $n!$ distribuciones iniciales posibles de los elementos sean igualmente probables. Sin embargo en muchas aplicaciones esta condición puede no ser realista. Si se utiliza un programa de ordenación para actualizar un archivo, por ejemplo, lo más probable es que se le pida ordenar matrices cuyos elementos ya estén casi ordenados, teniendo tan sólo unos pocos elementos nuevos que no estén es su sitio. En este caso, el comportamiento medio de casos seleccionados aleatoriamente será una guía muy mala en lo tocante a su rendimiento real.

Un análisis útil del comportamiento medio del algoritmo requiere por tanto un conocimiento *a priori* acerca de la distribución de los casos que hay que resolver. Este suele ser un requisito poco realista. Sobre todo cuando se utiliza un algoritmo como un procedimiento interno de algún algoritmo más complejo, puede no ser práctico estimar cuáles son los casos que aparecerán con mayor probabilidad, y cuáles se producirán tan sólo en raras ocasiones. Sin embargo, en la Sección 10.7 veremos cómo ciertos algoritmos pueden soslayar esta dificultad, y cómo se puede hacer que su comportamiento resulte independiente de los casos concretos que haya que resolver.

En lo que sigue tan sólo nos ocuparemos del análisis en el caso peor, a no ser que se indique lo contrario.

2.5 ¿QUÉ ES UNA OPERACIÓN ELEMENTAL?

Una *operación elemental* es aquella cuyo tiempo de ejecución se puede acotar superiormente por una constante que solamente dependerá de la implementación particular usada: de la máquina, del lenguaje de programación, etc. De esta manera la constante *no* depende ni del tamaño ni de los parámetros del ejemplar que se esté considerando. Dado que nos preocupan los tiempos de ejecución de algoritmos definidos salvo alguna constante multiplicativa, sólo el número de operaciones elementales ejecutadas importará en el análisis, y no el tiempo exacto requerido por cada una de ellas.

Por ejemplo, supongamos que cuando se analiza algún algoritmo, encontramos que para resolver un caso de un cierto tamaño se necesita efectuar a adiciones, m multiplicaciones, y s instrucciones de asignación. Supongamos que también se sabe que una suma nunca requiere más de t_a microsegundos, que una multiplicación nunca requiere más de t_m microsegundos, y que una asignación nunca requiere más de t_s microsegundos, donde t_a, t_m y t_s son constantes que dependen de la máquina utilizada. La suma, la multiplicación y la asignación pueden por tanto considerarse operaciones elementales. El tiempo total t requerido por nuestro algoritmo estará acotado por

$$t \leq at_a + mt_m + st_s$$
$$\leq \text{máx}\,(t_a, t_m, t_s) \times (a + m + s)$$

esto es, t está acotado por un múltiplo constante del número de operaciones elementales que hay que ejecutar.

Puesto que el tiempo exacto requerido por cada operación elemental no es importante, simplificaremos diciendo que las operaciones elementales se pueden ejecutar *a coste unitario.*

En la descripción del algoritmo, una única línea de programa se puede corresponder con un número variable de operaciones elementales. Por ejemplo, si T es una matriz de n elementos ($n > 0$), entonces el tiempo requerido para calcular

$$x \leftarrow min\,\{\, T[i] \mid 1 \leq i \leq n \,\}$$

crece con n, puesto que esto es una abreviatura de

```
x ← T[1]
para i ← 2 hasta n hacer
    si T[i] < x entonces x ← T[i]
```

De manera similar, algunas operaciones matemáticas son demasiado complejas para ser consideradas elementales. Si nos permitimos contar la evaluación de un factorial y una comprobación de divisibilidad como coste unitario, independientemente del tamaño de los operandos, el Teorema de Wilson, que afirma que el entero n para todo $n > 1$ divide $(n-1)! + 1$ si y sólo si n es primo, nos permitiría comprobar la primalidad de un entero con sorprendente eficiencia:

```
función Wilson (n)
    {Proporciona verdadero si y sólo si n es primo, n > 1}
    si n divide (n – 1)! + 1 entonces devolver verdadero
    si no devolver falso
```

El ejemplo del comienzo de esta sección sugería que podemos considerar que la adición (suma) y la multiplicación son operaciones de coste unitario, puesto que suponía que el tiempo requerido para estas operaciones podía acotarse mediante una constante. En teoría, sin embargo, estas operaciones no son elementales por cuanto el tiempo necesario para ejecutarlas aumenta con la longitud de los operandos. En la práctica, por otra parte, puede resultar sensato considerarlas como operaciones elementales con tal de que los operandos implicados sean del tamaño razonable en los ejemplares que esperemos encontrar. Hay dos ejemplos que ilustraran lo que queremos decir:

```
función Sum (n)
    {Calcule la suma de los enteros de 1 a n }
    sum ← 0
    para i ← 1 hasta n hacer sum ← sum + i
    devolver sum

función Fibonacci (n)
    {Calcula el n-ésimo término de la sucesión de Fibonacci;
    véase la Sección 1.6.4}
    i ← 1; j ← 0
    para k ← 1 hasta n hacer j ← i + j
                             i ← j – i
    devolver j
```

En el algoritmo llamado *Sum* el valor de *sum* sigue siendo razonable para todos los casos que uno puede de forma razonable esperar encontrarse en la práctica. Si estamos utilizando una máquina con palabras de 32 *bits*, todas las sumas se podrían ejecutar directamente siempre y cuando n no sea mayor que 65.535. En teoría, sin embargo, el algoritmo debería funcionar para *todos* los posibles valores de n. Ninguna máquina real puede de hecho ejecutar estas sumas con un coste unitario si seleccionamos un n suficientemente grande. El análisis del algoritmo debe por tanto depender del dominio de aplicación para el cual está destinado.

En el caso de *Fibonacci* la situación es distinta. Aquí basta con tomar $n = 47$ para hacer que la última adición «$j \leftarrow i + j$» dé lugar a un desbordamiento aritmético en máquinas de 32 *bits*. Para almacenar el resultado correspondiente a $n = 65.535$ se necesitarían 45.496 *bits*, lo cual es más de 1.420 palabras de computadora. Desde un punto de vista práctico, por tanto, no resulta realista considerar que estas sumas se pueden efectuar con un coste unitario. Más bien, debemos atribuirles un coste proporcional a la longitud de los operandos implicados. En la Sección 4.2.2 se muestra que este algoritmo requiere un tiempo cuadrático, aunque a primera vista su tiempo de ejecución parezca ser lineal.

En el caso de la multiplicación quizá sea razonable considerar que es una operación elemental para operandos suficientemente pequeños. Sin embargo, resulta más sencillo producir grandes operandos por multiplicación repetida que por adición, así que es todavía más importante asegurar que las operaciones aritméticas no se desborden. Además, cuando los operandos empiezan a volverse grandes, el tiempo requerido para efectuar una suma crece linealmente con el tamaño de los operandos, pero se cree que el tiempo requerido para efectuar una multiplicación crece más deprisa.

Puede surgir un problema similar cuando se analizan algoritmos que impliquen números reales, si es que la precisión requerida aumenta con el tamaño de los casos que haya que resolver.

Un ejemplo típico de este fenómeno es el uso de la fórmula de De Moivre (véase el problema 1.27) para calcular valores de la sucesión de Fibonacci. Esta fórmula nos dice que f_n, el n-ésimo término de la sucesión, es aproximadamente igual

a $\phi^n / \sqrt{5}$, en donde $\phi = (1 + \sqrt{5})/2$ es la *razón áurea*. La aproximación es suficientemente buena para que en principio se pueda obtener el valor exacto de f_n sin más que tomar el entero más próximo; véase el problema 2.23. Sin embargo, vimos más arriba que se requieren 45.496 *bits* para representar con precisión f_{65535}. Esto significa que sería preciso calcular la aproximación con el mismo grado de precisión para obtener la respuesta exacta. La aritmética ordinaria de coma flotante de simple o doble precisión, que emplea una o dos palabras de computadora, no sería, ciertamente, suficientemente precisa. En la mayoría de las situaciones prácticas, sin embargo, la utilización de aritmética de coma flotante de simple o doble precisión resulta ser satisfactoria, a pesar de la inevitable pérdida de precisión. Cuando esto sucede, es razonable tomar estas operaciones aritméticas como si fueran de coste unitario.

Para resumir, incluso la decisión de si una instrucción tan aparentemente inocua como «j← *i*+*j*» se puede considerar o no elemental requiere el uso de nuestro juicio. En lo que sigue, consideraremos que las sumas, restas, multiplicaciones, divisiones, operaciones de módulo, operaciones booleanas, comparaciones y asignaciones son operaciones elementales que se pueden ejecutar con un coste unitario a no ser que indiquemos explícitamente lo contrario.

2.6 ¿POR QUÉ HAY QUE BUSCAR LA EFICIENCIA?

A medida que las computadoras se van volviendo más y más rápidas puede parecer que apenas merece la pena invertir nuestro tiempo intentando diseñar unos algoritmos más eficientes. ¿No sería más sencillo esperar simplemente a la siguiente generación de computadoras? Los principios establecidos en las secciones anteriores muestran que esto no es verdad. Supongamos, para ilustrar el argumento, que para resolver un problema concreto disponemos de un algoritmo exponencial y de una computadora que puede ejecutar este algoritmo para casos de tamaño n en $10^{-4} \times 2^n$ segundos. El programa puede resolver un ejemplar de tamaño 10 en $10^{-4} \times 2^{10}$ segundos, o aproximadamente un décimo de segundo. Resolver un ejemplar de tamaño 20 requerirá aproximadamente un tiempo mil veces mayor, o aproximadamente dos minutos. Para resolver un ejemplar de tamaño 30 se necesitaría de nuevo mil veces más tiempo, así que no bastaría ni siquiera todo un día de tiempo de cálculo. Suponiendo que fuera posible hacer funcionar la computadora sin interrupción y sin errores durante un año, solamente se podría llegar a resolver un ejemplar de tamaño 38; véase el problema 2.15.

Supongamos que es necesario resolver ejemplares todavía más grandes que éste, y que con el dinero disponible puede uno permitirse comprar una computadora nueva cien veces más rápida que la primera. Ahora con el mismo algoritmo se puede resolver un ejemplar de tamaño n en sólo $10^{-6}\, 2^n$ segundos. Quizá tenga la sensación de haber desperdiciado el dinero, sin embargo, cuando determine que ahora, cuando tenga en marcha la nueva máquina durante todo un año, no se podrá ni siquiera resolver un ejemplar de tamaño 45. En general, si antes se podía resolver un ejemplar de tamaño n en un tiempo dado, la nueva máquina resolverá ejemplares de tamaño como mucho $n + \lg 100$, o aproximadamente $n + 7$, en el mismo tiempo.

Supongamos que en lugar de hacer esto decide invertir en algoritmia, y que, habiendo invertido la misma cantidad de dinero, se las arregla para encontrar un algoritmo cúbico para resolver el problema. Imagine, por ejemplo, que utilizando la máquina original y el algoritmo nuevo puede resolver un caso de tamaño n en $10^{-2} \times n^3$ segundos. Entonces para resolver un ejemplar de tamaño 10 necesitará 10 segundos, y un ejemplar de tamaño 20 seguirá requiriendo entre uno y dos minutos. Pero ahora un ejemplar de tamaño 30 se puede resolver en cuatro minutos y medio, y en un día se pueden resolver casos cuyo tamaño sea mayor que 200; con un año de cálculo se podría alcanzar casi el tamaño 1.500. Ilustramos esta tasa de crecimiento en la figura 2.1.

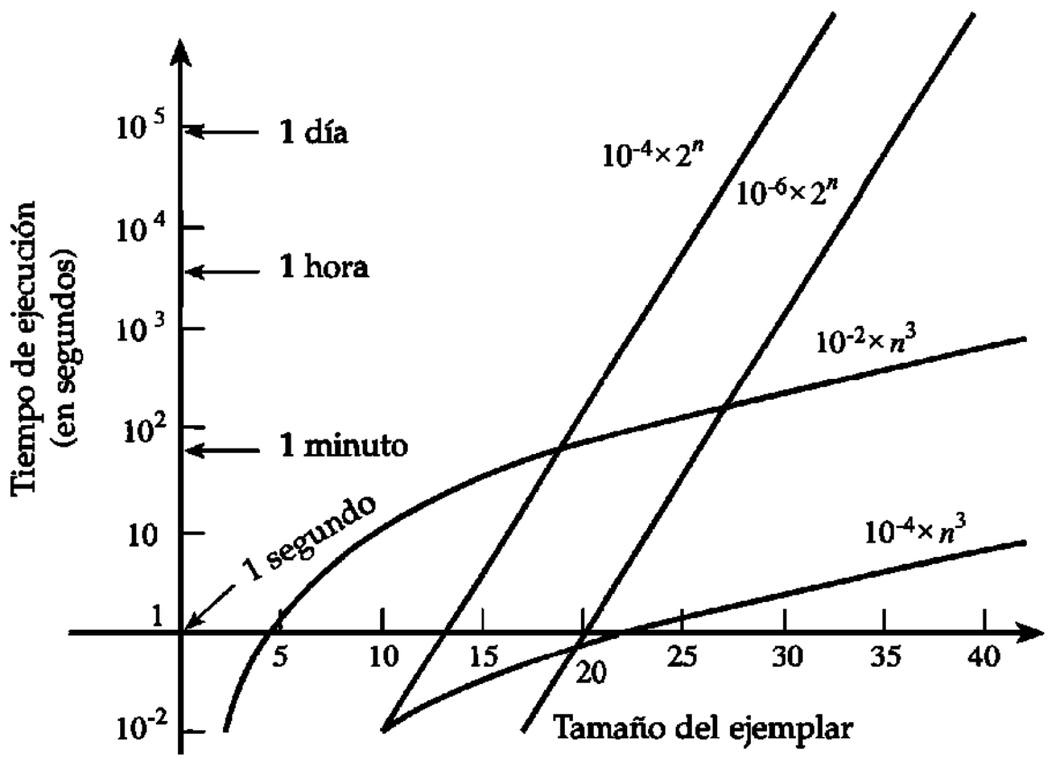


Figura 2.1. Algoritmia frente a hardware

El nuevo algoritmo no solamente ofrece una mejora mucho mayor que la adquisición del nuevo *hardware*, sino que además, suponiendo que pueda uno permitirse ambas cosas, hará que esta adquisición sea mucho más rentable. Si es posible utilizar tanto el algoritmo nuevo como una máquina cien veces más rápida que la anterior, entonces será posible resolver en el mismo tiempo ejemplares cuatro o cinco veces más grandes que sólo con el nuevo algoritmo nada más, y en la misma longitud de tiempo (el factor exacto es $\sqrt[3]{100}$). Compárese con la situación del algoritmo anterior, en la cual sólo se añadía 7 al tamaño del ejemplar; aquí se puede *multiplicar* el tamaño del ejemplar por cuatro o cinco. Sin embargo, el nuevo algoritmo no debería utilizarse alocadamente en todos los ejemplares del problema, en particular en los más pequeños. Hemos visto que en la máquina original el nuevo algoritmo requiere 10 segundos para re-

solver un ejemplar de tamaño 10, lo cual es cien veces *más lento* que el algoritmo viejo. El nuevo algoritmo solamente es más rápido para ejemplares de tamaño 20 o superior. Naturalmente, es posible combinar los dos algoritmos para formar un tercero que examinará el tamaño del ejemplar que hay que resolver antes de decidir qué método va a utilizar.

2.7 EJEMPLOS

Quizá se esté preguntando si es realmente posible en la práctica acelerar un algoritmo en la medida sugerida en la sección anterior. De hecho, se han dado casos en los cuales se han efectuado mejoras todavía más espectaculares, incluso para algoritmos ya muy establecidos.

2.7.1 Cálculo de determinantes

Los determinantes son importantes en álgebra lineal, y necesitamos saber la forma de calcularlos eficientemente. En nuestro contexto, proporcionan el mejor ejemplo de la diferencia que puede implicar un buen algoritmo, cuando se compara con otro algoritmo *clásico*. (Para disfrutar de esta sección no es preciso saber lo que es un determinante.)

Existen dos métodos bien conocidos para calcular determinantes. Uno está basado en la definición recursiva del determinante; el otro se denomina eliminación de Gauss-Jordan. El algoritmo recursivo requiere un tiempo proporcional a $n!$ para calcular el determinante de una matriz $n \times n$; véase el problema 1.31. Este crecimiento es incluso peor que el exponencial. En fuerte contraste, la eliminación de Gauss-Jordan requiere un tiempo proporcional a n^3 para la misma tarea.

Hemos programado ambos algoritmos en nuestra máquina local. El algoritmo de Gauss-Jordan halla el determinante de una matriz de 10×10 en una centésima de segundo; y necesita aproximadamente cinco segundos y medio en una matriz 100×100. Por otra parte, el algoritmo recursivo necesita más de 20 segundos para una sencilla matriz cinco por cinco, y 10 minutos para una matriz 10×10. Estimamos que el algoritmo recursivo necesitaría más de 10 millones de *años* para calcular el determinante de una matriz 20×20, ¡una tarea que es efectuada por el algoritmo de Gauss-Jordan en aproximadamente un veinteavo de segundo!

No se debe de concluir a partir de este ejemplo que los algoritmos recursivos son necesariamente malos. Al contrario, el Capítulo 7 describe una técnica en la cual la recursividad desempeña un papel fundamental en el diseño de algoritmos eficientes. En particular, un algoritmo recursivo que puede calcular el determinante de una matriz $n \times n$ en un tiempo proporcional a $n^{\lg 7}$, o aproximadamente $n^{2.81}$, se sigue del trabajo de Strassen, demostrando de esta manera que la eliminación de Gauss-Jordan no es óptima.

2.7.2 Ordenación

Los problemas de ordenación son de capital importancia en las ciencias de la computación, y en particular en la algoritmia. Se nos pide disponer por orden una colección de n objetos sobre los cuales está definida una *ordenación total*. Con esto queremos decir que cuando comparamos dos objetos cualesquiera de la colección, sabemos cuál de ellos debe ir en primer lugar. Para muchas clases de objetos este requisito es trivial: evidentemente 123 viene antes que 456 en el orden numérico, y el 1 de Agosto de 1991 va antes del 25 de Diciembre de 1995 por orden del calendario, así que tanto los enteros como las fechas están totalmente ordenados. Para otros objetos igualmente comunes, sin embargo, la definición de un orden total puede no ser tan sencilla. Por ejemplo, ¿cómo se ordenan dos números complejos? ¿Viene «general» antes o después de «General» por orden alfabético, o se trata de la misma palabra? (véase el problema 2. 16). Ninguna de estas preguntas tiene una respuesta evidente, pero mientras no se encuentre la respuesta los objetos correspondientes no se podrán ordenar.

Los problemas de ordenación suelen encontrarse dentro de algoritmos más complejos. Ya hemos visto dos algoritmos estándar de ordenación en la Sección 2.4: la ordenación por inserción y la ordenación por selección. Estos dos algoritmos, tal como vimos, requieren un tiempo cuadrático tanto en el caso peor como por término medio. Aun cuando ambos son excelentes cuando n es pequeño, existen otros algoritmos de ordenación que son más eficientes cuando n es grande. Entre otros podríamos utilizar el algoritmo ordenación por montículo *"heapsort"* de Willians (véase la Sección 5.7), ordenar por fusión *"mergesort"* (véase la Sección 7.4.1), o el algoritmo ordenación rápida *"quicksort"* de Hoare (véase la Sección 7.4.2). Todos estos algoritmos requieren un tiempo que está en el orden de $n \log n$ por término medio; los dos primeros requieren un tiempo del mismo orden incluso en el caso peor.

Para tener una idea más clara de las diferencias prácticas entre un tiempo que esté en el orden de n^2 y un tiempo que esté en el orden de $n \log n$, nosotros programamos el algoritmo de ordenación por inserción y *quicksort* en nuestra máquina local. La diferencia de eficiencia entre los dos algoritmos es marginal cuando es pequeño el número de elementos que hay que ordenar. *Quicksort* ya es casi dos veces más rápido que la inserción cuando se ordenan 50 elementos, y tres veces más rápido cuando se ordenan 100 elementos. Para ordenar 1.000 elementos, la inserción requiere más de tres segundos, mientras que *quicksort* requiere menos de un quinto de segundo. Cuando tenemos que ordenar 5.000 elementos, la ineficiencia de la ordenación por inserción se vuelve todavía más pronunciada: por término medio se necesita un minuto y medio, en comparación con poco más de un segundo para *quicksort*. En 30 segundos, *quicksort* puede manejar 10.000 elementos; estimamos que se necesitarían nueve horas y media para realizar el mismo trabajo utilizando la ordenación por inserción.

En el Capítulo 12 veremos que ningún algoritmo de ordenación que funcione por comparación de los elementos que hay que ordenar puede ser más rápido que el orden de $n \log n$, así que en este sentido *heapsort*, *mergesort* y *quicksort* son

lo más rápido que puede ser un algoritmo (aun cuando *quicksort* tiene un caso peor muy malo). Por supuesto los tiempos reales de ejecución dependen de las constantes multiplicativas ocultas en la definición de «el orden de». No obstante se pueden encontrar otros algoritmos de clasificación más rápidos, en casos especiales. Supongamos que los elementos que hay que ordenar son enteros de los cuales se saben que están entre 1 y 10.000. Entonces se puede utilizar el algoritmo siguiente.

```
procedimiento casilla(T[1..n ])
    {ordena enteros entre 1 y 10.000}
    matriz U[1 .. 10.000]
    para k ← 1 hasta 10.000 hacer U[k] ← 0
    para i ← 1 hasta n hacer
        k ← T[i]
        U[k] ← U[k] + 1
    i ← 0
    para k ← 1 hasta 10.000 hacer
        mientras U[k] ≠ 0 hacer
            i ← i + 1
            T[i] ← k
            U[k] ← U[k] - 1
```

Aquí U es una matriz de «casillas» para los elementos que hay que ordenar. Debe de haber una casilla distinta para todo posible elemento que se pudiera encontrar en T. El primer bucle vacía todas las casillas, y el segundo pone cada elemento de T en el lugar adecuado, y el tercero los vuelve a sacar otra vez por orden ascendente. Es fácil mostrar (véase el problema 2.17) que este algoritmo y sus variantes requieren un tiempo del orden de n. (La constante multiplicativa oculta depende de la cota superior al valor de los elementos que hay que ordenar, y que aquí es 10.000.) Por tanto cuando son aplicables estos algoritmos, son mejores que cualquier algoritmo que funcione comparando elementos; por otra parte, el requisito que debemos ser capaces de utilizar una casilla para toda posible clave significa que son aplicables con mucha menos frecuencia que los métodos generales de ordenación.

Los métodos paralelos de ordenación, que utilizan muchos procesadores para efectuar varias comparaciones simultáneamente, nos permiten ir todavía más deprisa. En el Capítulo 11 se esboza un ejemplo de algoritmo paralelo de ordenación.

2.7.3 Multiplicación de enteros muy grandes

Cuando un cálculo requiere manipular enteros muy grandes, puede suceder que los operandos se vuelvan demasiado largos para almacenarlos en una sola palabra de la computadora que se está utilizando. Por tanto tales operaciones dejan de ser elementales. Cuando sucede esto, se puede utilizar una representación tal como la de «doble precisión» de Fortran, o más generalmente, la aritmética de pre-

cisión múltiple. La mayoría de los lenguajes de programación orientados a objetos, tal como C ++ o Smalltalk, tienen clases predefinidas que hacen sencilla esta tarea. Ahora, sin embargo, debemos preguntarnos cómo aumenta el tiempo necesario para sumar, restar, multiplicar, o dividir dos enteros muy grandes con el tamaño de los operandos. Podemos medir este tamaño bien por el número de palabras de computadora necesarias para representar los operandos en una máquina o bien por la longitud de su representación en decimal o en binario. Puesto que estas medidas difieren sólo por una constante multiplicativa, esta opción no altera nuestro análisis del orden de eficiencia de los algoritmos en cuestión (pero véase el problema 2.18).

En esta sección consideraremos solamente la operación de multiplicación. El análisis de la suma y de la resta es mucho más sencillo, y se deja como el problema 2.19. Supóngase entonces que hemos de multiplicar dos enteros grandes de tamaño m y n respectivamente (¡no confunda el tamaño de los enteros con su valor!). El algoritmo clásico de la Sección 1.1 se puede adaptar fácilmente a este contexto. Vemos que multiplica cada cifra de uno de los operandos por la cifra del otro, y que ejecuta aproximadamente una adición elemental para cada una de estas multiplicaciones. (De hecho hay unas cuantas más, como consecuencia de los acarreos que se generan.) En una máquina multiplicamos cada palabra de uno de los operandos por cada palabra de la otra, y después hacemos aproximadamente una adición de longitud doble para cada una de estas multiplicaciones, pero el principio es exactamente el mismo. El tiempo requerido es por tanto del orden de mn. La multiplicación *à la russe* también requiere un tiempo del orden de mn, siempre y cuando pongamos el operando más pequeño en la columna de la izquierda y el mayor en la derecha; véase el problema 2.20. Por tanto no hay razón para preferirlo con respecto al algoritmo clásico, sobre todo porque es probable que la constante oculta sea mayor.

Existen algoritmos más eficientes para resolver el problema de la multiplicación de dos enteros muy grandes. El algoritmo divide y vencerás, que encontrábamos en la Sección 1.2, y que estudiaremos aún más en la Sección 7.1, requiere un tiempo del orden de $nm^{\lg(3/2)}$, o aproximadamente $nm^{0,59}$, en donde n es el tamaño del operando más grande y m el del más pequeño. Si ambos operandos son de tamaño n, el algoritmo requiere entonces un tiempo del orden de $n^{1,59}$, que es preferible al tiempo cuadrático que requieren tanto el algoritmo clásico como la multiplicación *à la russe*.

La diferencia entre el orden de n^2 y el orden de $n^{1,59}$ es menos espectacular que la existente entre el orden n^2 y el orden de $n \log n$, que veíamos en el caso de los algoritmos de ordenación. Para verificarlo hemos programado el algoritmo clásico y el algoritmo de divide y vencerás, y los hemos probado con operandos de diferentes tamaños. El algoritmo divide y vencerás, teóricamente mejor, ofrece una mejora real poco significativa en números de 600 dígitos: requiere unos 300 milisegundos, mientras que el algoritmo clásico requiere unos 400

milisegundos. Sin embargo, para operandos de longitud diez veces mayor, el algoritmo rápido es unas tres veces más eficiente que el algoritmo clásico: requieren aproximadamente 15 y 40 segundos, respectivamente. La ganancia de eficiencia sigue aumentando a medida que va ascendiendo el tamaño de los operandos.

Existen algoritmos más sofisticados, y el más rápido en la actualidad requiere un tiempo del orden de $n \log n \log \log n$ para multiplicar dos enteros de tamaño n. Sin embargo estos algoritmos más sofisticados son en su mayor parte de interés teórico; las constantes ocultas son tales que solamente llegan a ser competitivos para operandos *mucho* más grandes. Para casos «pequeños» que multipliquen operandos de unos pocos de miles de dígitos, resultan considerablemente más lentos que los algoritmos mencionados más arriba.

2.7.4 Cálculo del máximo común divisor

Sean m y n dos enteros positivos. El máximo común divisor de m y n, que se denota como mcd (m,n), es el mayor entero que divide exactamente tanto a m como a n. Cuando mcd $(m,n) = 1$, decimos que m y n son *primos entre sí*. Por ejemplo, mcd (10, 21) = 1 y mcd (6, 15) = 3, así que 10 y 21 son primos entre sí, pero 6 y 15 no lo son. El algoritmo evidente para calcular mcd (m,n) se obtiene directamente de la definición:

```
función mcd (m,n)
    i←min (m,n) + 1
    repetir i ← i – 1 hasta que i divide exactamente tanto a m como a n
    devolver i
```

El tiempo requerido por este algoritmo es del orden de la diferencia entre el menor de sus dos argumentos y su máximo común divisor. Cuando m y n son de tamaño similar y primos entre sí, requiere por tanto un tiempo del orden de n (Obsérvese que éste es el *valor* del operando, no su tamaño.)

Un algoritmo clásico para calcular mcd(m, n) consiste en factorizar primero m y n, y tomar entonces el producto de los factores primos comunes a m y n, elevando cada factor primo a la menor de sus potencias en los dos argumentos. Por ejemplo, para calcular mcd(120, 700) primero factorizamos $120 = 2^3 \times 5$ y $700 = 2^2 \times 5^2 \times 7$. Los factores comunes de 120 y 700 son por tanto 2 y 5, y sus potencias mínimas son 2 y 1 respectivamente. El máximo común divisor de 120 y 700 es por tanto $2^2 \times 5 = 20$.

Aun cuando este algoritmo es mejor que el dado anteriormente, nos exige factorizar m y n, una operación que nadie sabe realizar eficientemente cuando m y n son grandes; véase la Sección 10.7.4. De hecho, existe un algoritmo mucho más eficiente para calcular el máximo común divisor, conocido con el nombre de Algoritmo de Euclides, aun cuando su origen se puede retrotraer a bastante más atrás de los tiempos de los antiguos griegos:

```
función Euclides (m, n)
    mientras m > 0 hacer
        t←m
        m←n mod m
        n←t
    devolver n
```

Si consideramos que las operaciones aritméticas involucradas tienen coste unitario, este algoritmo requiere un tiempo del orden del logaritmo de sus argumentos —esto es, del orden de su tamaño— incluso en el caso peor; véase la Sección 4.4. Para ser exactos históricamente, el algoritmo original de Euclides funciona utilizando sustracciones sucesivas en lugar de calcular el módulo. En esta forma tiene más de 3.500 años de antigüedad.

2.7.5 Cálculo de la sucesión de Fibonacci

La sucesión de Fibonacci se presentó en la Sección 1.6.4. Recordamos al lector que los términos de esta secuencia se definen mediante la recurrencia siguiente:

$$\begin{cases} f_0 = 0;\ f_1 = 1 & \text{y} \\ f_n = f_{n-1} + f_{n-2} & \textit{para}\ \ n \geq 2 \end{cases}$$

Tal como vimos, la secuencia comienza en la forma 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... También vimos la fórmula de De Moivre

$$f_n = \frac{1}{\sqrt{5}}[\Phi^n - (-\Phi)^{-n}],$$

en donde $\Phi = (1 + \sqrt{5})/2$ es la *razón áurea*, e indicábamos que el término $(-\Phi)^{-n}$ se puede despreciar cuando n es grande. Por tanto el valor de f_n es del orden de n, y por tanto, el tamaño de f_n está en el orden de n. Sin embargo, la fórmula de De Moivre ofrece poca ayuda inmediata para calcular exactamente f_n, puesto que cuanto mayor se vuelve n, mayor es el grado de precisión que se requiere en los valores de $\sqrt{5}$ y de Φ; véase la Sección 2.5. En nuestra máquina local, un cálculo con simple precisión produce un error por primera vez cuando estamos calculando f_{66}.

El algoritmo recursivo que se obtiene directamente de la definición de la secuencia de Fibonacci se daba en la Sección 1.6.4 con el nombre *Fibonacci*:

```
función Fibrec (n)
    si n < 2 entonces devolver n
    sino devolver Fibrec (n − 1) + Fibrec (n − 2)
```

Este algoritmo es muy ineficiente porque recalcula muchas veces los mismos valores. Por ejemplo, para calcular los valores de *Fibrec* (5) necesitamos los valores de *Fibrec* (4) y de *Fibrec* (3); pero *Fibrec* (4) también requiere el cálculo *Fibrec* (3).

Es fácil comprobar que *Fibrec* (3) se calculará dos veces, *Fibrec* (2) tres veces, *Fibrec* (1) cinco veces y *Fibrec* (0) tres veces. (El número de llamadas a *Fibrec* (5), *Fibrec* (4), ... *Fibrec* (1), es por tanto 1, 1, 2, 3 y 5, respectivamente. No es coincidencia que éste sea el principio de la sucesión de Fibonacci; véase el problema 2.24.)

De hecho, el tiempo requerido para calcular f_n utilizando este algoritmo es del orden del propio valor de f_n, esto es, del orden de Φ^n. Para comprobarlo obsérvese que las llamadas recursivas solamente se detienen cuando *Fibrec* proporciona un valor 0 ó 1. La adición de estos resultados intermedios para obtener el resultado final f_n debe requerir al menos f_n operaciones, y por tanto ciertamente el algoritmo completo requiere un número de operaciones elementales que es por lo menos del orden de f_n. Esto se demostró formalmente mediante inducción constructiva en la Sección 1.6.4, junto con una demostración consistente en que el número de operaciones requerido no es mayor que el orden de f_n, siempre y cuando se cuenten las adiciones considerando que tienen un coste unitario. El caso en que las adiciones no se cuentan con un coste unitario llega a la misma conclusión, tal como se muestra mediante el análisis más preciso que se da en la Sección 4.2.3.

Para evitar calcular innecesariamente una y otra vez los mismos valores, es natural proceder como en la Sección 2.5, en la cual se presentaba un algoritmo *Fibonacci* distinto, algoritmo cuyo nombre cambiaremos por *Fibiter* en comparación con *Fibrec*:

```
función Fibiter (n)
    i ←1; j ← 0
    para k ← 1 hasta n hacer j← i + j
                             i← j – i
    devolver j
```

Este segundo algoritmo requiere un tiempo en el orden de n. Supongamos que se cuente cada adición como una operación elemental. La figura 2.2, que muestra algunos tiempos de computación que hemos observado en la práctica, ilustra la diferencia. Para evitar los problemas causados por operandos cada vez más largos, los cálculos mencionados en esta figura se efectuaron en módulo 10^7, lo cual quiere decir que solamente se calcularon las siete últimas cifras significativas de la respuesta. Los tiempos para *Fibrec* cuando $n \geq 50$ se estimaron utilizando el enfoque híbrido.

n	10	20	30	50	100
Fibrec	8 mseg	1 seg	2 min	21 días	10^9 años
Fibiter	$\frac{1}{6}$ mseg	$\frac{1}{3}$ mseg	$\frac{1}{2}$ mseg	$\frac{3}{4}$ mseg	$1\frac{1}{2}$ mseg

Figura 2.2. Comparación de algoritmos de Fibonacci módulo 10^7

Si no hacemos la suposición de que la suma es una operación elemental, *Fibiter* requiere un tiempo que es del orden de n^2, que sigue siendo mucho más rápido que el tiempo exponencial *Fibrec*. Sorprendentemente, existe un tercer algoritmo que proporciona una ventaja con respecto a *Fibiter* tan grande como *Fibiter* con respecto a *Fibrec*. Este tercer algoritmo requiere un tiempo del orden de logaritmo de n, siempre y cuando se consideren las operaciones aritméticas como de coste unitario. En caso contrario, el nuevo algoritmo sigue siendo más rápido que *Fibiter*, pero menos espectacularmente; véase el problema 7.33.

2.7.6 Transformada de Fourier

El algoritmo *Transformada Rápida de Fourier* es posiblemente el descubrimiento algorítmico que ha tenido un mayor impacto práctico en la historia. Las transformadas de Fourier son de importancia fundamental en aplicaciones tan dispares como la óptica, acústica, física cuántica, telecomunicaciones, teoría de sistemas y procesamiento de señales, incluyendo el reconocimiento del habla. Durante años, el avance en estas áreas de conocimiento ha estado limitado por el hecho de que los algoritmos conocidos para calcular las transformadas de Fourier requerían todos ellos demasiado tiempo.

El «descubrimiento» por parte de Cooley y Tukey en 1965 de un algoritmo rápido revolucionó la situación: los problemas que antes se consideraban inatacables podían ahora, por fin, ser abordados. En una de las primeras pruebas del «nuevo» algoritmo, se utilizó la transformada de Fourier para analizar los datos procedentes de un terremoto que se había producido en Alaska en 1964.

El algoritmo clásico requirió más de 26 minutos de computación; sin embargo, el algoritmo «nuevo» fue capaz de realizar la misma tarea en menos de dos segundos y medio.

Irónicamente, resultó que ya había sido publicado un algoritmo eficiente en 1942 por Danielson y Lanczos, y que toda la base teórica del algoritmo de Danielson y Lanczos había sido publicada por Runge y König en 1924. ¡Por si *esto* no fuera suficiente, Gauss describe un algoritmo similar en un artículo escrito alrededor de 1805 y publicado póstumamente en 1866!

2.8 ¿CUÁNDO QUEDA ESPECIFICADO UN ALGORITMO?

Al principio del libro decíamos que la ejecución de un algoritmo no debe normalmente implicar ninguna decisión subjetiva, ni tampoco debe hacer necesario el uso de la intuición o de la creatividad; posteriormente, decíamos que uno casi siempre debe estar contento si consigue demostrar que los algoritmos son correctos en abstracto, ignorando las limitaciones prácticas, y, todavía más tarde, proponíamos considerar que la mayoría de las operaciones aritméticas son elementales, a no ser que se indique explícitamente lo contrario.

Todo esto está muy bien, ¿pero qué deberíamos hacer si las consideraciones prácticas nos obligan a abandonar esta postura tan cómoda, y a tener en cuenta las limitaciones de la maquinaria disponible? Por ejemplo, todo algoritmo que deba

calcular el valor exacto de f_{100} se verá obligado a considerar que ciertas operaciones aritméticas —ciertamente la suma y posiblemente la multiplicación también (véase el problema 7.33)— no son elementales (recuérdese que f_{100} es un número con 21 dígitos decimales). Con toda probabilidad esto se tendrá en cuenta empleando un paquete de programas que permita realizar operaciones aritméticas con enteros muy grandes. Si no especificamos exactamente cómo debe implementar el paquete la operación de precisión múltiple, entonces la elección del método que hay que utilizar se puede considerar una decisión subjetiva, y el algoritmo propuesto estará especificado de forma incompleta. ¿Es esto importante?

La respuesta es que sí, en ciertos casos es importante. En partes posteriores del libro nos encontraremos con algoritmos cuyo rendimiento dependerá del método que se utilice para multiplicar enteros grandes. Para tales algoritmos (y hablando formalmente, para *todo* algoritmo) no basta simplemente con escribir una instrucción como $x \leftarrow y \times z$, dejando que el lector seleccione cualquier técnica que tenga a mano para implementar esta multiplicación. Para implementar completamente el algoritmo, también debemos especificar la forma en que deben ser implementadas las operaciones aritméticas necesarias.

Para hacer más sencillas las cosas, sin embargo, seguiremos utilizando la palabra *algoritmo* para ciertas descripciones incompletas de esta clase. Los detalles se rellenarán posteriormente, si es que nuestros análisis los requieren.

2.9 PROBLEMAS

Problema 2.1. Hallar un algoritmo más práctico para calcular la fecha de Pascua que el que se ha dado en el problema 1.2. ¿Cuál será la fecha de Pascua en el año 2000? ¿Cuál es el dominio de definición de su algoritmo?

Problema 2.2. En el juego de ajedrez, las piezas y los movimientos del adversario son todos ellos visibles. Decimos que el ajedrez es un juego con *información completa*. En juegos tales como el bridge o el póker, sin embargo, no se sabe cómo se han repartido las cartas.

¿Hace esto imposible definir un algoritmo para jugar bien al bridge o al póker? ¿Sería uno de estos algoritmos necesariamente probabilístico?

¿Qué pasa con el *backgammon*? Aquí se pueden ver las piezas y los movimientos del adversario, pero no es posible predecir cuál será el resultado de los dados en tiradas futuras. ¿Altera esto la situación?

Problema 2.3. A veces se afirma que en la actualidad el *hardware* es tan barato y la mano de obra es tan cara que nunca merece la pena desperdiciar el tiempo de un programador para quitarle unos pocos segundos a la ejecución de un programa. ¿Significa esto que la algoritmia está destinada a ser un estudio teórico de interés solamente formal, sin aplicaciones prácticas? Justifique su respuesta.

Problema 2.4. Utilizando la técnica llamada «memoria virtual», es posible liberar al programador de la mayoría de sus preocupaciones en lo tocante al tamaño real del espacio disponible en su máquina. ¿Significa esto que la cantidad de almacenamiento utilizada por un algoritmo nunca tiene interés en la práctica? Justifique su respuesta.

Problema 2.5. Supongamos que se mide el rendimiento de un programa, utilizando quizás algún seguimiento de la ejecución, y que entonces se optimizan las partes más uti-

lizadas del código. Sin embargo, se tiene buen cuidado de no modificar el algoritmo subyacente. ¿Espera usted obtener (a) una ganancia de eficiencia por un factor constante, sea cual fuere el problema resuelto, o (b) una ganancia de eficiencia que se vaya volviendo proporcionalmente mayor a medida que aumente el tamaño del problema? Justifique su respuesta.

Problema 2.6. Un algoritmo de ordenación requiere un segundo para ordenar 1.000 elementos en su máquina local. ¿Cuánto tiempo esperaría usted que requiriera para clasificar 10.000 elementos (a) si cree que el algoritmo requiere un tiempo aproximadamente proporcional a n^2 y (b) si cree que el algoritmo requiere un tiempo aproximadamente proporcional a $n \log n$?

Problema 2.7. Dos algoritmos requieren n^2 días y n^3 segundos respectivamente para resolver un caso de tamaño n. Demostrar que sólo en aquellos casos que requieran más de veinte millones de años para su resolución, el algoritmo cuadrático tendrá un mejor rendimiento que el algoritmo cúbico.

Problema 2.8. Dos algoritmos requieren n^2 días y n^3 segundos respectivamente para resolver casos de tamaño n. ¿Cuál es el tamaño del caso más pequeño en el cual el primer algoritmo es más rápido que el segundo? ¿Aproximadamente, cuánto tiempo tarda en resolverse este caso?

Problema 2.9. Simular los algoritmos de clasificación por inserción y de ordenación por selección dados en la Sección 2.4 aplicándolos a las dos matrices siguientes: U = [1, 2, 3, 4, 5, 6] y V = [6, 5, 4, 3, 2, 1]. ¿Funciona la clasificación por inserción más deprisa en la matriz U o en la matriz V? ¿Qué sucede con el algoritmo de selección? Justifique sus respuestas.

Problema 2. 10. Suponga que desea «ordenar» una matriz W = [1, 1, 1, 1, 1, 1], con todos sus elementos iguales, utilizando (a) ordenación por inserción y (b) clasificación por selección. Establecer una comparación con la ordenación de las matrices U y V del problema anterior.

Problema 2. 11. Se le pide clasificar un archivo que contiene enteros entre 0 y 999.999. No puede utilizar un millón de casillas, así que en su lugar decide utilizar mil casillas numeradas desde 0 a 999. Comienza la clasificación colocando cada entero en la casilla correspondiente a sus tres primeras cifras. A continuación utiliza *mil veces* la ordenación por inserción para ordenar el contenido de cada casilla por separado. Y por último se vacían las casillas por orden para obtener una secuencia totalmente ordenada.

¿Esperaría usted que esta técnica sea más rápida, más lenta o equivalente a utilizar simplemente la clasificación por inserción aplicada a toda la secuencia?: (a) por término medio, y (b) en el caso peor. Justifique sus respuestas.

Problema 2.12. ¿Es razonable, desde un punto de vista práctico, considerar la división como operación elemental? (a) siempre, (b) algunas veces, o (c) nunca. Justifique su respuesta. Si le parece necesario puede tratar la división de enteros y la división de números reales de forma separada.

Problema 2.13. Supongamos que n es una variable entera en un programa que estamos escribiendo. Considérese la instrucción $x \leftarrow \sin(n)$ donde se puede suponer que n está en radianes. Desde un punto de vista práctico, ¿consideraría usted la ejecución de esta instrucción como una operación elemental? (a) siempre, (b) algunas veces, o (c) nunca. Justifique su respuesta. ¿Y qué pasa con la instrucción $x \leftarrow \sin(n\pi)$?

Problema 2.14. En la Sección 2.5 veíamos que el teorema de Wilson se podía utilizar para comprobar la primalidad de cualquier número en un tiempo constante si los factoriales y las comprobaciones de divisibilidad entera se tomaban como de coste unitario, independientemente del tamaño de los números implicados. Claramente, esto no sería razonable.

Utilice el teorema de Wilson junto con el teorema binomial de Newton para diseñar un algoritmo capaz de decidir en un tiempo del orden de logaritmo de n si un entero es o no primo, siempre y cuando las adiciones, las multiplicaciones y las comprobaciones de divisibilidad por enteros se consideren como de coste unitario, pero los factoriales y las exponenciales no. El punto fundamental de este ejercicio *no* es proporcionar un algoritmo útil, sino demostrar que no es razonable considerar que las multiplicaciones son operaciones elementales en general.

Problema 2.15. Un cierto algoritmo requiere $10^{-4} \times 2^n$ segundos para resolver un caso de tamaño n. Mostrar que en un año podría apenas resolver un caso de tamaño 38. ¿Qué tamaño de caso se podría resolver en un año en una máquina que fuera 100 veces más rápida?

Un segundo algoritmo requiere $10^{-2} \times n^3$ segundos para resolver un caso de tamaño n ¿Qué tamaño de caso podría resolver en un año? ¿Qué tamaño de caso se podría resolver en un año en una máquina cien veces más rápida?

Mostrar que, sin embargo, el segundo algoritmo es más lento que el primero para casos de tamaño menor que 20.

Problema 2.16. Supongamos por el momento que una palabra se define como una cadena de letras sin espacios ni signos de puntuación intermedios, de tal forma que «can't» puede considerar como dos palabras, y «can't-a-box» como cuatro[5]. Necesitamos clasificar un archivo de palabras como éstas. Diseñar un algoritmo que, dadas dos cadenas de caracteres cualesquiera decida si representan o no la misma palabra y en caso contrario, cuál de ellas debe ir primero. ¿Qué haría su algoritmo con (a) «Mackay» y «MacKay», y (b) con «anchorage» y «Anchorage»? Todo algoritmo no debería poner «Mackay» y «Anchorage» antes de «aardvark» ni después de «zymurgy».

Como problema mucho más difícil, diseñar un algoritmo para comparar las entradas de la guía de teléfonos. Este algoritmo deberá enfrentarse a todo tipo de cadenas extrañas, incluyendo signos de puntuación («E-Z limpiezas»), números («A-1 Farmacias»), acentos y otros signos diacríticos («Adèle Núñez»), y abreviaturas («St Catherine St. Tavern»). ¿Cómo lo hace la compañía telefónica?

Problema 2.17. Mostrar que la clasificación por casillas requiere un tiempo que es del orden de n para clasificar n elementos que estén entre dos límites.

Problema 2.18. En la Sección 2.7.3 decíamos que el análisis de algoritmos para enteros grandes no se ve afectado por la elección de una medida para el tamaño de los operandos: el número de palabras de computadora que necesitan, o la longitud de su representación en decimal o en binario funcionarán igualmente bien. Mostrar que este comentario sería realmente falso si estuviéramos considerando algoritmos de tiempo exponencial.

Problema 2.19. ¿Cuánto tiempo se requiere para sumar o restar dos enteros grandes de tamaños m y n respectivamente? Esboce el algoritmo adecuado.

Problema 2.20. ¿Cuánto tiempo se requiere para multiplicar dos enteros grandes de tamaños m y n, respectivamente, empleando la

multiplicación *á la russe*? (a) si el operando más pequeño está en la columna de la izquierda, y (b) si el operando más grande está en la columna de la izquierda? Por supuesto, no se deben de tomar la adición, la multiplicación por dos y la división por dos como operaciones elementales en este problema.

Problema 2.21. ¿Cuánto tiempo se requiere para multiplicar dos enteros grandes de tamaños m y n respectivamente, utilizando la multiplicación en torno a un rectángulo? (véase el problema 1.9).

Problema 2.22. Calcúlese mcd(606, 979) (a) por factorización de 606 y 979, y seleccionando los factores comunes de potencia adecuada, y (*b*) utilizando el algoritmo de Euclides.

Problema 2.23. Utilizar la fórmula de Moivre para f_n para demostrar que f_n es el entero más próximo a $\phi^n / \sqrt{5}$ para todo $n \geq 1$.

Problema 2.24. Mostrar que cuando se está calculando f_n utilizando *Fibrec* de la Sección 2.7.5 existe un total de f_{n+1-i} llamadas de *Fibrec* (*i*) para $i = 1,2,\ldots, n$ y f_{n-1} llamadas a *Fibrec* (0).

Problema 2.25. Sea $g(n)$ el número de formas en que se puede escribir una cadena de n ceros y unos de tal forma que nunca haya dos ceros consecutivos. Por ejemplo, cuando $n = 1$ las cadenas posibles son 0 y 1, así que $g(1) = 2$; cuando $n = 2$ las cadenas posibles son 01,11 y 10, así que $g(2) = 3$; cuando $n = 3$ las cadenas posibles son 010, 011, 101, 110 y 111, así que $g(3) = 5$. Demostrar que $g(n) = f_{n+2}$.

2.10. REFERENCIAS Y TEXTOS MÁS AVANZADOS

Para reforzar nuestros comentarios acerca de la importancia de diseñar algoritmos eficientes, animamos al lector a que examine Bentley (1984), ¡que ofrece demostraciones experimentales de que la utilización inteligente de la algorítmica ¡puede permitir a un TRS-80 funcionar más deprisa que un CRAY-1!

De los trabajos de Strassen (1969) y Bunch y Hopcroft (1974) se sigue un algoritmo capaz de calcular un determinante $n \times n$ en un tiempo que es del orden de $n^{2,81}$. En este libro se discuten varios algoritmos de clasificación; Knuth (1973) es mucho más exhaustivo acerca de este tema. El algoritmo «divide y vencerás» que puede multiplicar números de n-cifras en un tiempo del orden de $n^{1,59}$ es atribuido a Karatsuba y Ofman (1962); se describe con detalle en el Capítulo 7. Un algoritmo más rápido para la multiplicación de enteros muy grandes debido a Schönhage y Strassen (1971) se ejecuta en un tiempo que es del orden de «n log n log log n»; véase Brassard, Monet y Zuffellato (1986) para más detalles. El algoritmo de Euclides se puede encontrar en el libro VII de los *Elementos* de Euclides: véase Heath (1926). El algoritmo rápido para calcular la sucesión de Fibonacci al que se hace mención en la Sección 2.7.5 es explicado en Gries y Levin (1980) y Urbanek (1980); véanse también Brassard y Bratley (1988) para una competición entre este algoritmo rápido y los que se dan en la Sección 2.7.5.

El primer algoritmo publicado para calcular eficientemente transformadas discretas de Fourier es el de Danielson y Lanczos (1942). Estos autores mencionan que la fuente de su método se retrotrae hasta Runge y König (1924). A la vista de la gran importancia de las transformadas de Fourier es sorprendente que la existencia de un algoritmo rápido permaneciera casi en el anonimato hasta su redescubrimiento prácticamente un cuarto de siglo después por parte de Cooley y Tukey (1965). Para una descripción más completa de la historia de la transformada rápida de Fourier, se puede leer Cooley, Lewis y Welch (1967). Se ofrecen más detalles y aplicaciones a la multiplicación rápida de enteros y a la aritmética simbólica de polinomios en Brassard y Bratley (1988).

Se puede encontrar una solución del problema 2.1 atribuida a Gauss bien en Larousse (1968) o en Kirkerud (1989); véase también el *Devocionario anglicano*, disponible en cualquier iglesia anglicana. La solución del problema 2.14 ha sido dada por Shamir (1979).

Capítulo 3

Notación asintótica

3.1 INTRODUCCIÓN

Un aspecto importante de este libro es el que concierne a la determinación de la eficiencia de algoritmos. En la Sección 2.3, veíamos que este conocimiento puede ayudarnos, por ejemplo, a elegir uno de entre varios algoritmos en pugna. Recuérdese que deseamos determinar matemáticamente la cantidad de recursos que necesita el algoritmo como función del tamaño (o a veces del valor) de los casos considerados. Dado que no existe una computadora estándar con la cual se puedan comparar todas las medidas de *tiempo de ejecución*, vimos también en la Sección 2.3 que nos contentaremos con expresar el tiempo requerido por el algoritmo salvo por una constante multiplicativa. Con este objetivo, presentamos ahora formalmente la *notación asintótica* que se utiliza a lo largo de todo el libro. Además, esta notación permite realizar simplificaciones sustanciales aun cuando estemos interesados en medir algo más tangible que el tiempo de ejecución, tal como el número de veces que se ejecuta una instrucción dada dentro de un programa.

Esta notación se denomina «asintótica» porque trata acerca del comportamiento de funciones en el límite, esto es, para valores suficientemente grandes de su parámetro. En consecuencia, los argumentos basados en la notación asintótica pueden no llegar a tener un valor práctico cuando el parámetro adopta valores «de la vida real». Sin embargo, las enseñanzas de la notación asintótica suelen tener una relevancia significativa. Esto se debe a que, como regla del pulgar, un algoritmo que sea superior asintóticamente suele ser (aunque no siempre) preferible incluso en casos de tamaño moderado.

3.2 UNA NOTACIÓN PARA "EL ORDEN DE"

Sea $t: \mathbb{N} \to \mathbb{R}^{\geq 0}$ una función arbitraria de los números naturales en los números reales no negativos, tal como $t(n) = 27n^2 + \frac{355}{113}n + 12$. Se puede pensar que n repre-

senta el tamaño del ejemplar sobre el cual es preciso que se aplique un algoritmo dado, y en $t(n)$ como representante de la cantidad de un recurso dado que se invierte en ese ejemplar por una implementación particular de este algoritmo. Por ejemplo, podría ser que la implementación invirtiera $t(n)$ microsegundos en el ejemplar peor en un caso de tamaño n, o quizá $t(n)$ represente la cantidad de espacio. Tal como se ha visto, la función $t(n)$ puede muy bien depender de la implementación más que depender únicamente del algoritmo. Recuerde sin embargo el principio de invariancia, que dice que la *razón* de los tiempos de ejecución de dos implementaciones diferentes del mismo algoritmo siempre está acotada por encima y por debajo mediante constantes predeterminadas. (Las constantes pueden depender de las implementaciones pero no del tamaño del ejemplar.)

Considérese ahora una función $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ tal como $f(n) = n^2$. Diremos que $t(n)$ está en el *orden de* $f(n)$ si $t(n)$ está acotada superiormente por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande. Matemáticamente, esto significa que existe una constante real y positiva c y un entero *umbral* n_0 tal que $t(n) \leq cf(n)$, siempre que $n \geq n_0$.

Por ejemplo, considérense las funciones $f(n)$ y $t(n)$ definidas anteriormente. Está claro que tanto $n \leq n^2$ como $1 \leq n^2$, siempre que $n \geq 1$. Por tanto, siempre y cuando $n \geq 1$:

$$t(n) = 27n^2 + \frac{355}{113}n + 12$$

$$\geq 27n^2 + \frac{355}{113}n^2 + 12n^2$$

$$= 42\frac{16}{113}n^2 = 42\frac{16}{113}f(n)$$

Tomando $c = 42\frac{16}{113}$ (o cualquier valor mayor) y $n_0 = 1$, concluiremos por tanto que $t(n)$ es del orden de $f(n)$ por cuanto $t(n) \leq cf(n)$ siempre que $n \geq n_0$. Resulta fácil ver que también podríamos haber seleccionado $c = 28$ y $n_0 = 6$. Esta compensación entre el valor más pequeño posible para n_0 y el de c es bastante frecuente.

De esta manera, si la implementación de un algoritmo requiere en el caso peor $27n^2 + \frac{355}{113}n + 12$ microsegundos para resolver un caso de tamaño n, podríamos simplificar diciendo que el tiempo está en el orden de n^2. Naturalmente, no tiene sentido afirmar que estamos hablando del orden de n^2 *microsegundos*, por cuanto esta cantidad solamente se diferencia por un factor constante, digamos, de n^2 años. Hay algo más importante: «el orden de n^2» caracteriza no solamente el requerido por una implementación particular del algoritmo, sino también (por el principio de invariancia) el tiempo requerido por *cualquier* implementación. Por tanto, tenemos derecho a afirmar que el *algoritmo en sí* requiere un tiempo que está en el orden de n^2, o más sencillamente, que requiere un *tiempo cuadrático*.

Es conveniente disponer de un símbolo matemático para representar *el orden de*. Una vez más sea $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ una función arbitraria de los números naturales en los reales no negativos. Le indicará mediante $O(f(n))$ —que se lee «O de $f(n)$»— *el conjunto* de todas las funciones $t: \mathbb{N} \to \mathbb{R}^{\geq 0}$ tales que $t(n) \leq cf(n)$ para todo $n \geq n_0$ para una constante positiva real c y para un umbral entero n_0. En otras palabras:

$$O(f(n)) = \{t: \mathbb{N} \to \mathbb{R}^{\geq 0} \mid (\exists c \in \mathbb{R}^+)\ (\forall n \in \mathbb{N})[t(n) \leq cf(n)]\}.$$

Aun cuando es natural utilizar el símbolo «$\in$» de teoría de conjuntos para denotar que n^2 es del orden de n^3 tal como en «$n^2 \in O(n^3)$», le advertimos que la notación tradicional es $n^2 = O(n^3)$. Por tanto, no se sorprenda si encuentra una de estas «igualdades» de un solo sentido, porque uno nunca escribiría $O(n^3) = n^2$, en otros libros o artículos científicos. Aquellos que utilizan las igualdades de un solo sentido dicen que n^2 es del orden de (o algunas veces *sobre* el orden de) n^3, o bien que n^2 es $O(n^3)$. Otra diferencia significativa que se puede encontrar en la definición de la notación O es que algunos escritores permiten que $O(f(n))$ incluya las funciones de los números naturales en el conjunto de todos los enteros reales —incluyendo los reales negativos— y definen que una función está en $O(f(n))$ si su *valor absoluto* está en lo que nosotros llamamos $O(f(n))$.

Por comodidad, nos permitiremos utilizar incorrectamente la notación de vez en cuando (así como otra notación que se presenta más adelante en este capítulo). Por ejemplo, podemos decir que $t(n)$ está en el orden de $f(n)$ incluso si $t(n)$ es negativo o no está definido para un número finito de valores de n. De manera similar, podemos hablar acerca del orden de $f(n)$ incluso en el caso de que $f(n)$ sea negativa o no esté definida para un número finito de valores de n. Diremos entonces que $t(n) \in O(f(n))$ si existe una constante real y positiva c y un umbral entero n_0 tales que tanto $t(n)$ como $f(n)$ están bien definidos y $0 \leq t(n) \leq cf(n)$ siempre que $n \geq n_0$, independientemente de lo que suceda en estas funciones cuando $n < n_0$. Por ejemplo, está permitido hablar del orden de $n/\log n$, aun cuando esta función no está definida cuando $n = 0$ ó $n = 1$, y es correcto escribir $n^3 - 3n^2 - n - 8 \in O(n^3)$ aun cuando $n^3 - 3n^2 - n - 8 < 0$ cuando $n \leq 3$.

El umbral de n_0 suele resultar útil para simplificar argumentos, pero nunca es necesario cuando consideramos funciones *estrictamente* positivas. Sean $f, t: \mathbb{N}^+ \to \mathbb{R}^+$ dos funciones de los números naturales en los reales estrictamente positivos. La **regla del umbral** afirma que $t(n) \in O(f(n))$ si y sólo si existe una constante real positiva tal que $t(n) \in O(f(n))$ para *cada* número natural n. La parte *si* de la regla es evidente, puesto que toda propiedad que sea verdadera para cada número natural será también verdadera para cada entero suficientemente grande (basta con tomar $n_0 = 0$ como umbral). Supongamos para la parte *solo si* que $t(n) \in O(f(n))$. Sean c y n_0 las constantes relevantes, de tal modo que $t(n) \leq cf(n)$ siempre que $n \geq n_0$. Supongamos $n_0 > 0$, puesto que en caso contrario no hay nada que demostrar. Sea $b = \max\{t(n)/f(n) \mid 0 \leq n < n_0\}$ el mayor valor tomado por la razón de t y f sobre los números naturales menores que n_0; esta definición tiene sentido precisamente

porque $f(n)$ no puede ser cero y $n_0 > 0$. Por definición del máximo, $b \geq t(n)/f(n)$, y por tanto $t(n) \leq bf(n)$, siempre que $0 \leq n < n_0$. Ya sabemos que $t(n) \leq cf(n)$ siempre que $n \geq n_0$. Por tanto $t(n) \leq af(n)$ para cada número natural n, tal como teníamos que demostrar, siempre y cuando seleccionemos un a que sea al menos tan grande como b y c, como por ejemplo $a = \text{máx}(b,c)$. La regla del umbral se puede generalizar para decir que si $t(n) \in O(f(n))$ y si $f(n)$ es estrictamente positiva para todos los $n \geq n_0$ para algún n_0, entonces se puede utilizar este n_0 como umbral para la notación O: existe una constante real positiva c tal que $t(n) \leq cf(n)$ para todo $n \geq n_0$.

Una regla útil para demostrar que una función es del orden de otra es la **regla del máximo**. Sean $f, g: \mathbb{N} \to \mathbb{R}^{\geq 0}$ dos funciones arbitrarias de los números naturales en los reales no negativos. La regla del máximo dice que $O(f(n) + g(n)) = O(\text{máx}(f(n),g(n)))$. Más específicamente, sean $p, q: \mathbb{N} \to \mathbb{R}^{\geq 0}$ definidas para todo número natural n mediante $p(n) = f(n) + g(n)$ y $q(n) = \max(f(n), g(n))$, y consideremos una función arbitraria $t: \mathbb{N} \to \mathbb{R}^{\geq 0}$. La regla del máximo dice que $t(n) \in O(p(n))$ si y sólo si $t(n) \in O(q(n))$. Esta regla se puede generalizar a cualquier número constante finito de funciones. Antes de demostrarlo, ilustramos una interpretación natural de esta regla. Considérese un algoritmo que se realiza en tres pasos: inicialización, procesamiento y finalización. Supongamos para este argumento que estos pasos requieren un tiempo de $O(n^2)$, $O(n^3)$ y $O(n \log n)$, respectivamente. Queda por tanto claro (véase la Sección 4.2) que el algoritmo completo requiere un tiempo de $O(n^2+n^3+ n \log n)$. Aun cuando no sería difícil demostrar directamente que este orden es el mismo que $O(n^3)$, resulta inmediato de la regla de máximo:

$$\begin{aligned} O(n^2 + n^3 + n \log n) &= O(\text{máx}(n^2,n^3, n \log n)) \\ &= O(n^3) \end{aligned}$$

En otras palabras, aun cuando el tiempo requerido por el algoritmo es lógicamente la suma de los tiempos requeridos por sus partes separadas, dicho tiempo es del orden del tiempo requerido por la parte que consuma más tiempo, siempre y cuando el número de partes sea una constante, independientemente del tamaño de la entrada.

Ahora demostraremos la regla del máximo para el caso de dos funciones. El caso general de cualquier número predeterminado de funciones se deja como ejercicio. Obsérvese que

$$f(n) + g(n) = \text{mín}(f(n),g(n)) + \text{máx}(f(n),g(n))$$

y

$$0 \leq \text{mín}(f(n),g(n)) \leq \text{máx}(f(n), g(n))$$

Se sigue entonces que

$$\text{máx}(f(n),g(n)) \leq f(n) + g(n) \leq 2 \text{ máx}(f(n), g(n)) \tag{3.1}$$

Considérese ahora cualquier $t(n) \in O(f(n) + g(n))$. Sea c una constante adecuada tal que $t(n) \leq c(f(n)+ g(n))$ para todo n suficientemente grande. Por la ecuación 3.1 se

sigue que $t(n) \leq 2c$ máx $(f(n), g(n))$. Por tanto, $t(n)$ está acotado superiormente por un múltiplo real y positivo (a saber, $2c$) de máx $(f(n), g(n))$ para todo n suficientemente grande, lo cual demuestra que $t(n) \in O$ (máx $(f(n), g(n))$). Para el sentido inverso, considérese cualquier $t(n) \in O$ (máx$(f(n), g(n))$). Sea $\hat{c}$ una nueva constante adecuada tal que $t(n) \leq \hat{c}$ máx$(f(n), g(n))$ para n suficientemente grande. Una vez más por la ecuación 3.1 se sigue que $t(n) \leq \hat{c}$ $(f(n) + g(n))$. Por definición de la notación O esto implica directamente que $t(n) \in O(f(n) + g(n))$, lo cual completa la demostración de la regla del máximo. De acuerdo con nuestra utilización incorrecta pero permitida de la notación en algunas ocasiones, tenemos derecho a invocar la regla del máximo aun cuando las funciones implicadas sean negativas o bien no estén definidas en un conjunto finito de valores. Tenga cuidado de no utilizar la regla, sin embargo, si algunas de las funciones son negativas con infinita frecuencia; en caso contrario corre el riesgo de razonar en la forma siguiente:

$$O(n) = O(n + n^2 - n^2) = O\,(\text{máx}(n, n^2, -n^2)) = O(n^2)$$

en donde la igualdad intermedia se obtiene por el uso incorrecto de la regla del máximo.

La regla del máximo nos dice que si $t(n)$ es una función complicada tal como $t(n) = 12n^3 \log n - 5n^2 + \log^2 n + 36$ y si $f(n)$ es el término más significativo de $t(n)$ descartando el coeficiente, aquí $f(n) = n^3 \log n$, entonces $O(t(n)) = O(f(n))$, lo cual permite una simplificación dramática pero automática en la notación asintótica. En otras palabras, se pueden despreciar los términos de orden inferior porque son despreciables en comparación con el término de orden superior para n suficientemente grande. Obsérvese en este caso que uno *no* debería demostrar $O(t(n)) = O(f(n))$ razonando del modo siguiente:

$$\begin{aligned} O(t(n)) &= O\,(\text{máx}(12n^3 \log n, -5n^2, \log^2 n + 36)) \\ &= O\,(12n^3 \log n) = O\,(n^3 \log n) \end{aligned}$$

donde la segunda línea se obtiene a partir de la regla del máximo. Éste no utiliza adecuadamente la regla del máximo por cuanto la función $-5n^2$ es negativa. Sin embargo, el razonamiento siguiente es correcto:

$$\begin{aligned} O(t(n)) &= O(11n^3 \log n + n^3 \log n, -5n^2 + \log^2 n + 36) \\ &= O(\text{máx}(11n^3 \log n, n^3 \log n, -5n^2 + \log^2 n, 36)) \\ &= O(11n^3 \log n) = O(n^3 \log n) \end{aligned}$$

Aun cuando $n^3 \log n - 5n^2$ sea negativo y 36 sea mayor que $11n^3 \log n$ para valores pequeños de n, todo va bien, puesto que esto no sucede para valores de n suficientemente grandes.

Otra observación útil es que resulta normalmente innecesario especificar la base del logaritmo dentro de la notación asintótica. Esto se debe a que el $\log_a n = \log_a b \times \log_b n$ para todos los reales positivos a, b y n tales que ni a ni b es igual a 1. Lo

que importa es que el $\log_a b$ es una constante positiva, cuando a y b son constantes mayores que 1. Por tanto el $\log_a n$ y el $\log_b n$ solamente difieren en una constante multiplicativa. A partir de esto, resulta elemental demostrar que $O(\log_a n) = O(\log_b n)$, lo cual normalmente simplificaremos en la forma $O(\log n)$. Esta observación también se aplica a funciones más complicadas, tales como los polinomios positivos que impliquen a n y a $\log n$, y a los cocientes de tales polinomios. Por ejemplo, $O(n \lg n)$ es lo mismo que el más habitual $O(n \log n)$, y $O(n^2/(\log_3 n\sqrt{n} \lg n))$ es lo mismo que $O((n/\log n)^{1.5})$. Sin embargo, la base del logaritmo *no* se puede ignorar cuando es más pequeña que 1, cuando no es una constante, tal como en $O(\log_{1/2} n) \neq O(\log n)$, o cuando el logaritmo se encuentra en el exponente, tal como en $O(2^{\lg n}) \neq O(2^{\log n})$.

Es fácil demostrar que la notación «$\in O$» es reflexiva y transitiva. En otras palabras, $f(n) \in O(f(n))$ para toda función $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ y para cualesquiera funciones $f, g, h: \mathbb{N} \to \mathbb{R}^{\geq 0}$ se cumple que si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$ entonces $f(n) \in O(h(n))$; véanse los problemas 3.9 y 3.10. Como resultado, esta notación proporciona una forma de definir una ordenación parcial en el conjunto de las funciones, y consiguientemente en el conjunto eficiencias relativas de los diferentes algoritmos para resolver un problema dado; véase el problema 3.21. Sin embargo, el orden inducido no es total por cuanto existen funciones $f, g: \mathbb{N} \to \mathbb{R}^{\geq 0}$ tales que ni $f(n) \in O(g(n))$ ni $g(n) \in O(f(n))$; véase el problema 3.11.

Hemos visto varios ejemplos de funciones $t(n)$ y $f(n)$ para las cuales es sencillo demostrar que $t(n) \in O(f(n))$. Para éstas, basta encontrar las constantes c y n_0 adecuadas y mostrar que es válida la relación deseada.

¿Cómo podríamos demostrar que una función dada $t(n)$ *no* pertenece al orden de otra función $f(n)$? La forma más sencilla es utilizar una demostración por contradicción. El ejemplo siguiente ilustra esta circunstancia. Sea $t(n) = \frac{1}{1000} n^3$ y $f(n) = 1000n^2$. Si se prueban varios valores de n menores que un millón, se observa que $t(n) < f(n)$, lo cual puede llevarnos a pensar por inducción que $t(n) \in O(f(n))$, tomando uno como constante multiplicativa. Si se intenta demostrar esto, sin embargo, es probable que acabe uno con la siguiente demostración por contradicción que es falsa. Para demostrar que $t(n) \notin O(f(n))$, supóngase para demostrar la contradicción que $t(n) \in O(f(n))$. Utilizando la regla del umbral generalizada, esto implica la existencia de una constante real y positiva c tal que $t(n) \leq cf(n)$ para todos los $n \geq 1$. Pero $t(n) \leq cf(n)$ significa que $\frac{1}{1000} n^3 \leq 1000\, cn^2$, lo cual implica que $n \leq c10^6$. En otras palabras, al suponer que $t(n) \in O(f(n))$ estamos demostrando que todo entero positivo n es menor que alguna constante predeterminada, lo cual es evidentemente falso.

La herramienta más potente y versátil para demostrar que algunas funciones están en el orden de otras y para demostrar lo contrario es la **regla del límite**, la cual manifiesta que dadas las funciones arbitrarias f y g: $\mathbb{N} \to \mathbb{R}^{\geq 0}$:

1. Si $\lim \frac{f(n)}{g(n)} \in \mathbb{R}^+$ entonces $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$

2. Si $\lim \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ pero $g(n) \notin O(f(n))$, y

3. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $f(n) \notin O(g(n))$ pero $g(n) \in O(f(n))$

Ilustraremos la utilización de esta regla antes de demostrarla. Considérense las dos funciones $f(n) = \log n$ y $g(n) = \sqrt{n}$. Deseamos determinar el orden relativo de estas funciones. Puesto que tanto $f(n)$ como $g(n)$ tienden a infinito cuando n tiende a infinito, utilizaremos la regla de l'Hôpital para calcular

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{1/n}{1/(2\sqrt{n})}$$

$$= \lim_{n\to\infty} 2/\sqrt{n} = 0$$

Ahora bien, la regla del límite demuestra inmediatamente que $\log n \in O(\sqrt{n})$ mientras que $\sqrt{n} \notin O(\log n)$. En otras palabras, $\sqrt{n}$ crece asintóticamente más deprisa que $\log n$. Demostraremos a continuación la regla del límite.

1. Supóngase que $\lim_{n\to\infty} f(n) / g(n) = \ell \in \mathbb{R}^+$. Sea $\delta = \ell$ y $c = 2\ell$. Por definición del límite, la diferencia en valor absoluto entre $f(n)/g(n)$ y ℓ no es mayor que δ para todo n suficientemente grande. Pero esto significa que $f(n)/g(n) \leq \ell + \delta = c$. Por tanto, hemos mostrado una constante real y positiva c tal que $f(n) \leq cg(n)$ para todo n suficientemente grande, lo cual demuestra que $f(n) \in O(g(n))$. El hecho consistente en que $g(n) \in O(f(n))$ es automático, puesto que $\lim_{n\to\infty} f(n) / g(n) = \ell \in \mathbb{R}^+$ implica que $\lim_{n\to\infty} g(n) / f(n) = 1/\ell \in \mathbb{R}^+$ y por tanto el razomiento anterior es aplicable *mutatis mutandis* intercambiando $f(n)$ y $g(n)$.

2. Supongamos que el $\lim_{n\to\infty} f(n) / g(n) = 0$. Se fija arbitrariamente el valor $\delta = 1$. Por definición del límite, el valor absoluto de $f(n) / g(n)$ no es mayor que δ para todo n suficientemente grande. Por tanto $f(n)/g(n) \leq \delta$, lo cual implica que $f(n) \leq g(n)$ puesto que $\delta = 1$, para todo n suficientemente grande. Esto demuestra que $f(n) \in O(g(n))$. Para demostrar que $g(n) \notin O(f(n))$ supóngase para demostrar por contradicción que $g(n) \in O(f(n))$. Esta suposición implica la existencia de una constante real positiva c tal que

$$g(n) \leq cf(n)$$

y por tanto $1/c \leq f(n)/g(n)$, para todo n suficientemente grande. Dado que

$$\lim_{n\to\infty} f(n) / g(n)$$

existe por la suposición consistente en que es igual a 0, y dado que $\lim_{n\to\infty} 1/c = 1/c$ existe también, la proposición 1.7.8 es aplicable, y concluiremos que $1/c \leq \lim_{n\to\infty} f(n) / g(n) = 0$, lo cual es una contradicción puesto que $c > 0$. Hemos contradicho la suposición consistente en que $g(n) \in O(f(n))$ y por tanto hemos establecido, según se quería demostrar, que $g(n) \notin O(f(n))$.

3. Supóngase que $\lim_{n \to \infty} f(n) / g(n) = +\infty$. Esto implica que

$$\lim_{n \to \infty} g(n) / f(n) = 0$$

y por tanto el caso anterior es aplicable *mutatis mutandis* después de intercambiar $f(n)$ y $g(n)$.

La inversa de la regla del límite no es necesariamente válida: no siempre sucede que el $\lim_{n \to \infty} f(n) / g(n) \in \mathbb{R}^+$ cuando $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$. Aun cuando ciertamente se sigue que el límite es estrictamente positivo, si existe, el problema es que puede no existir. Considérese por ejemplo $f(n) = n$ y $g(n) = 2^{\lfloor \lg n \rfloor}$. Es fácil ver que $g(n) \leq f(n) \leq 2g(n)$ para todo $n \geq 1$, y por tanto $f(n)$ y $g(n)$ son ambos del orden del otro. Sin embargo, es igualmente sencillo ver que $f(n) / g(n)$ oscila entre 1 y 2, y por tanto no existe el límite de esta razón.

3.3 OTRA NOTACIÓN ASINTÓTICA

La notación Omega. Considérese el problema de clasificación que se discutía en la Sección 2.7.2. Vimos que los algoritmos de ordenación más evidentes, tal como la ordenación por inserción y la ordenación por selección requieren un tiempo en $O(n^2)$, mientras que los algoritmos más sofisticados como *ordenar-montículo* son más eficientes porque la efectúan en un tiempo del orden de $O(n \log n)$. Pero resulta sencillo mostrar que $n \log n \in O(n^2)$. Como resultado, ¡*es correcto* decir que *ordenar-montículo* tiene un tiempo de $O(n^2)$, o incluso de $O(n^3)$ si vamos a ello! Esto resulta confuso en principio, pero es la consecuencia inevitable del hecho consistente en que la notación O solamente se ha diseñado para proporcionar cotas *superiores* sobre la cantidad de recursos requeridos. Claramente, necesitamos una notación dual para cotas *inferiores*. Esto es la notación Ω.

Considérense una vez más dos funciones $t, f: \quad \to \mathbb{R}^{\geq 0}$ de los números naturales en los números reales no negativos. Diremos que $t(n)$ está en Omega de $f(n)$, lo cual se denota como $t(n) \in \Omega(f(n))$, si $t(n)$ está acotada *inferiormente* por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande. Matemáticamente, esto significa que existe una constante real positiva d y un umbral entero n_0 tal que $t(n) \geq df(n)$ siempre que $n \geq n_0$:

$$\Omega(f(n)) = \{t: \mathbb{N} \to \mathbb{R}^{\geq 0} \mid (\exists\, d \in \mathbb{R}^+)\ (\overset{\infty}{\forall}\, n \in \mathbb{N})\ [t(n) \geq df(n)]\}$$

Es fácil ver la **regla de dualidad**: $t(n) \in \Omega(f(n))$ si y sólo si $f(n) \in O(t(n))$ porque $t(n) \geq df(n)$ si y sólo si $f(n) \leq \frac{1}{d}\, t(n)$. Por tanto, puede uno cuestionarse la utilidad de presentar una notación para Ω cuando parece que la notación O tenga la misma potencia expresiva. La razón es que resulta más natural decir que un algoritmo requiere un tiempo en Ω de n^2 que utilizar el equivalente matemático más oscuro «n^2 está en O del tiempo tardado por el algoritmo».

Gracias a la regla de dualidad, sabemos a partir de la sección anterior que $\sqrt{n} \in \Omega(\log n)$ mientras que $\log n \notin \Omega(\sqrt{n})$, entre otros muchos ejemplos. Hay algo más importante, y es que la regla de dualidad se puede utilizar de la forma evidente para transformar la regla del límite, la regla del máximo y la regla del umbral en reglas acerca de la notación Ω.

A pesar de la fuerte similitud entre las notaciones O y Ω, existe un aspecto en el cual falla su dualidad. Recuérdese que lo que más frecuentemente nos interesa de los algoritmos es la eficiencia en el *caso peor*. Por tanto, cuando decimos que una implementación del algoritmo requiere $t(n)$ microsegundos, queremos decir que $t(n)$ es el tiempo máximo que requiere esa implementación para todos los casos de tamaño n. Sea un $f(n)$ tal que $t(n) \in O(f(n))$. Esto significa que existe una constante real positiva c tal que $t(n) \leq cf(n)$ para todo n suficientemente grande. Dado que ningún ejemplar de tamaño n puede tomar más tiempo que el tiempo máximo requerido por los ejemplares de ese tamaño, se sigue que la implementación requiere un tiempo acotado por $cf(n)$ microsegundos para todos los ejemplares suficientemente grandes. Suponiendo que solamente exista un número finito de ejemplares de tamaño, puede haber solamente un número finito de ejemplares, todos ellos de tamaño menor que el umbral, sobre los cuales la implementación requiera un tiempo mayor que $cf(n)$ microsegundos. Suponiendo que $f(n)$ nunca sea cero, todos éstos se pueden resolver utilizando una constante multiplicativa mayor, tal como se hace en la demostración de la regla del umbral.

Por contraste, supongamos que $t(n) \in \Omega(f(n))$. Una vez más, esto significa que existe una constante real positiva d tal que $t(n) \geq df(n)$ para todo n suficientemente grande. Ahora bien, dado que $t(n)$ denota el comportamiento de esa implementación en el caso peor, podemos inferir solamente que, para cada n suficientemente grande, existe al menos *un ejemplar* de tamaño n tal que la implementación requiere al menos $df(n)$ microsegundos para ese ejemplo. Esto no excluye la posibilidad de un comportamiento mucho más rápido sobre otros ejemplares del mismo tamaño. Por tanto, pueden existir infinitos ejemplares en los cuales la implementación requiera menos de $df(n)$ microsegundos. La ordenación por inserción ofrece un ejemplo típico de este comportamiento. Vimos en la Sección 2.4 que se requiere un tiempo cuadrático en el caso peor, pero sin embargo existen infinitos ejemplares en los cuales se ejecuta en un tiempo lineal. Por tanto, tenemos derecho a manifestar que el tiempo de ejecución en el caso peor está tanto en $O(n^2)$ como en $\Omega(n^2)$. Sin embargo, la primera afirmación dice que todo caso suficientemente grande se podrá clasificar en un tiempo cuadrático, mientras que la segunda se limita a decir que al menos un ejemplar de cada tamaño suficientemente grande requiere realmente el tiempo cuadrático: el algoritmo puede ser más rápido en otros ejemplares del mismo tamaño.

Algunos autores definen la notación Ω en una forma que es diferente de forma sutil pero importante. Dicen que $t(n) \in \Omega(f(n))$ si existe una constante real y positiva d tal que $t(n) \geq df(n)$ para un *número infinito* de valores de n, mientras que nosotros requerimos que esta relación sea válida para todos los valores de n salvo un

número finito. Con esta definición, un algoritmo que requiera un tiempo en $\Omega(f(n))$ en el caso peor es tal que existen infinitos casos en los cuales requiere al menos $df(n)$ microsegundos para la constante real positiva adecuada d. Esto se corresponde de forma más próxima a nuestra idea intuitiva de lo que debería ser una cota inferior para el rendimiento de un algoritmo. Resulta más natural que lo que *nosotros* queremos decir con «requerir un tiempo en Ω de $f(n)$». Sin embargo, preferimos nuestra definición porque es más fácil trabajar con ella. En particular, la definición modificada de Ω no es transitiva y la regla de dualidad falla.

En este libro, utilizamos la notación Ω sobre todo para proporcionar cotas inferiores acerca del tiempo de ejecución (u otros recursos) de los algoritmos. Sin embargo, esta notación suele utilizarse para proporcionar cotas inferiores acerca de la dificultad intrínseca de resolver ciertos problemas. Por ejemplo, veremos en la Sección 12.2.1 que *cualquier* algoritmo que ordene con éxito n elementos debe requerir un tiempo en $\Omega(n \log n)$ siempre y cuando la única operación efectuada sobre los elementos que hay que ordenar consista en compararlos por parejas para determinar si son iguales, y, en caso contrario, cuál de ellos es el mayor; como resultado, diremos que el problema consistente en ordenar mediante comparaciones tiene una *complejidad* de tiempo de ejecución que está en $\Omega(n \log n)$. En general, resulta mucho más difícil determinar la complejidad de un problema que determinar una cota inferior sobre el tiempo de ejecución de un cierto algoritmo que lo resuelva. Hablaremos más sobre este tema en el Capítulo 12.

La notación Theta. Cuando analizamos el comportamiento de un algoritmo, nos sentimos especialmente felices si su tiempo de ejecución está acotado tanto por encima como por debajo mediante múltiplos reales positivos posiblemente distintos de una misma función. Por esta razón, presentamos la notación Θ. Diremos que $t(n)$ está en Theta de $f(n)$, o lo que es lo mismo que $t(n)$ está en el *orden exacto* de $f(n)$, y lo denotamos $t(n) \in \Theta(f(n))$, si $t(n)$ pertenece tanto a $O(f(n))$ como a $\Omega(f(n))$. La definición formal de Θ es:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

Esto es equivalente a decir que $\Theta(f(n))$ es

$$\{t: \mathbb{N} \to \mathbb{R}^{\geq 0} \mid (\exists c,d \in \mathbb{R}^{+})(\forall\ n \in \mathbb{N})\ [df(n) \leq t(n) \leq cf(n)]\}$$

La regla del umbral y la regla del máximo, que formulábamos en el contexto de la notación O, es aplicable *mutatis mutandis* a la notación Θ. Curiosamente, la regla del límite para la notación Θ se reformula en la forma siguiente. Considérense las funciones arbitrarias f y $g: \mathbb{N} \to \mathbb{R}^{\geq 0}$.

1. Si $\lim \frac{f(n)}{g(n)} \in \mathbb{R}^{+}$ entonces $f(n) \in \Theta(g(n))$

2. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ pero $f(n) \notin \Theta(g(n))$ y

3. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $f(n) \in \Omega(g(n))$ pero $f(n) \notin \Theta(g(n))$

Como ejercicio de manipulación de la notación asintótica, demostraremos ahora un hecho útil:

$$\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$$

para cualquier entero $k \geq 0$, en donde la suma del lado izquierdo se considera como función de n. Por supuesto, éste hecho se sigue de forma inmediata de la proposición 1.7.16, pero resulta instructivo demostrarlo directamente.

La dirección «O» es fácil de probar. Para esto, obsérvese simplemente que $i^k \leq n^k$ siempre que $1 \leq i \leq n$. Por tanto, $\sum_{i=1}^{n} i^k \leq \sum_{i=1}^{n} n^k = n^{k+1}$ para todo $n \geq 1$, lo cual demuestra que $\sum_{i=1}^{n} i^k \in O(n^{k+1})$ utilizando 1 como constante multiplicativa.

Para demostrar la dirección «Ω», obsérvese que $i^k \geq (n/2)^k$ siempre que $i \geq \lceil n/2 \rceil$ y que el número de enteros que hay entre $\lceil n/2 \rceil$ y n, ambos inclusive, es mayor que $n/2$. Por tanto, siempre que $n \geq 1$ (lo cual implica que $\lceil n/2 \rceil \geq 1$):

$$\sum_{i=1}^{n} i^k \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} i^k \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \left(\frac{n}{2}\right)^k \geq \frac{n}{2} \times \left(\frac{n}{2}\right)^k = \frac{n^{k+1}}{2^{k+1}}$$

Esto demuestra que $\sum_{i=1}^{n} i^k \in \Omega(n^{k+1})$ utilizando $1/2^{k+1}$ como constante multiplicativa. Se obtiene una constante más ajustada en el problema 3.23.

3.4 NOTACIÓN ASINTÓTICA CONDICIONAL

Hay muchos algoritmos que resultan más fáciles de analizar si en un principio limitamos nuestra atención a aquellos casos cuyo tamaño satisfaga una cierta condición, tal como ser una potencia de 2. Considérese por ejemplo el algoritmo «divide y vencerás» para multiplicar enteros grandes que veíamos en la Sección 1.2. Sea n el tamaño de los enteros que hay que multiplicar (suponíamos que eran de un mismo tamaño). El algoritmo se ejecuta directamente si $n = 1$, lo cual requiere a microsegundos para una constante apropiada a. Si $n > 1$, el algoritmo se ejecuta recursivamente multiplicando cuatro parejas de enteros de tamaño $\lceil n/2 \rceil$ (o tres parejas en el algoritmo mejorado que estudiaremos en el Capítulo 7). Además, es precisa una cantidad lineal de tiempo para efectuar tareas adicionales. Por sencillez, digamos que el trabajo adicional requiere de n microsegundos para una constante adecuada b (para ser exactos, sería necesario un tiempo entre $b_1 n$ y $b_2 n$ microsegundos para las constantes adecuadas b_1 y b_2 (hablaremos más acerca de esto en la Sección 4.7.6).

El tiempo que requiere este algoritmo está dado consiguientemente por la función $t: \mathbb{N} \to \mathbb{R}^{\geq 0}$ que se define recursivamente en la forma

$$t(n) = \begin{cases} a & \text{si } n = 1 \\ 4t(\lceil n/2 \rceil) + bn & \text{en caso contrario} \end{cases} \tag{3.2}$$

En la Sección 4.7 estudiaremos técnicas para resolver recurrencias, pero desafortunadamente la ecuación 3.2 no se puede manejar directamente mediante esas técnicas porque la función $\lceil n/2 \rceil$ es dificultosa. Sin embargo, nuestra recurrencia es fácil de resolver siempre y cuando consideremos solamente el caso en el cual n es una potencia de 2: en este caso $\lceil n/2 \rceil = n/2$ y el redondeo por exceso desaparece. Las técnicas de la Sección 4.7 producen

$$t(n) = (a+b)n^2 - bn$$

siempre y cuando «n» sea una potencia de 2. Dado que el término de orden inferior «$-bn$» se puede despreciar, se sigue que $t(n)$ es del orden exacto de n^2, siempre y cuando, insistimos, n sea una potencia de 2. Esto se denota mediante $t(n) \in \Theta(n^2 \mid n$ es una potencia de 2).

Más generalmente, sean $f, t: \mathbb{N} \to \mathbb{R}^{\geq 0}$ dos funciones de los números naturales en los números reales no negativos, y sea $P: \mathbb{N} \to \{verdadero, falso\}$ una propiedad de los enteros. Decimos que $t(n)$ está en $O(f(n) \mid P(n))$ si $t(n)$ está acotado superiormente por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande tal que $P(n)$ sea válido. Formalmente, $O(f(n) \mid P(n))$ se define en la forma

$$\{t: \mathbb{N} \to \mathbb{R}^{\geq 0} \mid (\exists c \in \mathbb{R}^+)\ (\overset{\infty}{\forall} n \in \mathbb{N})\ [P(n) \Rightarrow t(n) \leq cf(n)]\}$$

Los conjuntos $\Omega(f(n) \mid P(n))$ y $\Theta(f(n) \mid P(n))$ se definen de forma similar. Abusando de la notación en una forma familiar escribiremos que $t(n) \in O(f(n) \mid P(n))$ incluso si $t(n)$ y $f(n)$ son negativas o no están definidas en un cierto número de puntos arbitrario —quizá infinito— de n en los cuales $P(n)$ no es válida.

La notación asintótica condicional es algo más que una mera comodidad de notación: su interés principal es que se puede eliminar, en general, una vez que ha sido utilizada para facilitar el análisis de un algoritmo. Para verlo, necesitamos unas cuantas definiciones. Una función $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ es *asintóticamente no decreciente* si existe un umbral entero n_0 tal que $f(n) \leq f(n+1)$ para todo $n \geq n_0$. Esto implica por inducción matemática que $f(n) \leq f(m)$ siempre que $m \geq n \geq n_0$. Sea $b \geq 2$ cualquier entero. Se dice que la función f es b-uniforme[1], si, además de ser asintóticamente no decreciente, satisface la condición $f(bn) \in O(f(n))$. En otras palabras, tiene que existir una constante c (dependiente de b) tal que $f(bn) \leq cf(n)$ para todo $n \geq n_0$. (No hay pérdida de generalidad al utilizar un mismo umbral n_0 para los dos propósitos.) Una función se dice *uniforme* o de *ajuste* si es b-uniforme para todo entero $b \geq 2$.

La mayoría de las funciones que se suele uno encontrar en el análisis de algoritmos son uniformes (de ajuste, suaves), tal como $\log n$, $n \log n$, n^2 o cualquier polinomio cuyo primer coeficiente sea positivo. Sin embargo, las funciones que crecen demasiado deprisa tal como $n^{\lg n}$, 2^n o $n!$, no son suaves porque la razón $f(2n)/f(n)$ no está acotada. Por ejemplo:

$$(2n)^{\lg(2n)} = 2n^2\, n^{\lg n}$$

lo cual muestra que $(2n)^{\lg(2n)} \notin O(n^{\lg n})$ porque $2n^2$ no se puede acotar superiormente mediante una constante. Por otra parte, las funciones que están acotadas superiormente mediante algún polinomio, suelen ser suaves siempre que sean asintóticamente no decrecientes; e incluso si no son asintóticamente no decrecientes existen buenas posibilidades de que estén en el orden exacto de alguna otra función que sea suave. Por ejemplo, supongamos que $b(n)$ denota el número de *bits* iguales a uno en la expansión binaria de n (tal como $b(13) = 3$ porque 13 se escribe 1101 en binario) y consideremos $f(n) = b(n) + \lg n$. Es fácil ver que $f(n)$ no es eventualmente no decreciente —y por tanto, que no es uniforme— porque $b(2^k - 1) = k$ mientras que $b(2^k) = 1$ para todo k. Sin embargo, $f(n) \in \Theta(\log n)$, que es una función uniforme. (Este ejemplo no es tan artificial como pudiera parecer; véase la Sección 7.8.) En raras ocasiones se encontrarán funciones de crecimiento lento que no estén en el orden exacto de una función suave.

Una propiedad útil de la uniformidad (ajuste o suavización) es que si f es b-uniforme para algún entero concreto $b \geq 2$, entonces, de hecho, es uniforme. Para demostrar esto, considérense dos enteros cualesquiera a y b mayores o iguales que 2. Supongamos que f es b-uniforme. Debemos demostrar que f es también a-uniforme. Sean c y n_0 constantes tales que $f(bn) \leq c\, f(n)$ y $f(n) \leq f(n+1)$ para todo $n \geq n_0$. Sea $i = \lceil \log_b a \rceil$. Por definición del logaritmo, $a = b^{\log_b a} \leq b^{\lceil \log_b a \rceil} = b^i$. Considérse cualquier $n \geq 0$. Es fácil mostrar por inducción matemática sobre la b-uniformidad de f que $f(b^i n) \leq c^i f(n)$. Pero $f(an) \leq f(b^i n)$ (porque f es asintomáticamente no decreciente y $b^i n \geq an \geq n_0$. Se sigue que $f(an) \leq \hat{c}\, f(n)$ para $\hat{c} = c^i$, y por tanto f es a-uniforme.

Las funciones uniformes son interesantes como consecuencia de la **regla de la uniformidad** (ajuste o suavidad). Sea f: $\mathbb{N}^+ \to \mathbb{R}^{\geq 0}$ una función suave y sea t: $\mathbb{N}^+ \to \mathbb{N}^+$ una función asintóticamente no decreciente. Considérese cualquier entero $b \geq 2$. La regla de uniformidad afirma que $t(n) \in \Theta(f(n))$ siempre que $t(n) \in \Theta(f(n) \mid n$ es una potencia de $b)$. La regla es aplicable igualmente a la notación O y a la notación Ω. Antes de demostrar la regla, la ilustraremos con el ejemplo utilizado al principio de esta sección.

Ya vimos que es fácil obtener la fórmula condicional asintótica

$$t(n) \in \Theta(n^2 \mid n \text{ es una potencia de } 2) \tag{3.3}$$

a partir de la ecuación 3.2, mientras que resulta más difícil efectuar el análisis de $t(n)$ cuando n no es una potencia de 2. La regla de la uniformidad nos permite in-

ferir directamente de la ecuación 3.3 que $t(n) \in \Theta(n^2)$ siempre y cuando especifiquemos que n^2 es una función suave y que $t(n)$ es asintóticamente no decreciente. La primera condición es inmediata, por cuanto n^2 es obviamente no decreciente y $(2n)^2 = 4n^2$. La segunda se demuestra fácilmente por inducción matemática sobre la ecuación 3.2; véase el problema 3.28. Por tanto, el uso de la notación condicional asintótica como paso intermedio da lugar al resultado final consistente en que $t(n) \in \Theta(n^2)$ incondicionalmente.

Demostraremos ahora la regla de uniformidad. Sea $f(n)$ una función uniforme y sea $t(n)$ una función eventualmente no decreciente tal que $t(n) \in \Theta(f(n) \mid n$ es una potencia de $b)$ para algún entero $b \geq 2$. Sea n_0 el mayor de los umbrales implicado por las condiciones anteriores: $f(m) \leq f(m+1)$, $t(m) \leq t(m+1)$ y $f(bm) \leq cf(m)$ siempre que $m \geq n_0$ y $df(m) \leq t(m) \leq af(m)$ siempre que $m \geq n_0$ sea una potencia de b, para constantes adecuadas a,c y d. Para todo entero positivo n, sea $\underline{n}$ la mayor potencia de b que no es mayor que n (formalmente, $\underline{n} = b^{\lfloor \log_b n \rfloor}$) y sea $\overline{n} = b\underline{n}$. Por definición, $n/b < \underline{n} \leq n < \overline{n}$ y $\overline{n}$ es una potencia de b. Considérese cualquier $n \geq \max(1, bn_0)$.

$$t(n) \leq t(\overline{n}) \leq af(\overline{n}) = af(b\underline{n}) \leq acf(\underline{n}) \leq acf(n)$$

Esta ecuación emplea sucesivamente los hechos consistentes en que t es eventualmente no decreciente (y $\overline{n} \geq n \geq n_0$), $t(m)$ es del orden de $f(m)$ cuando m es una potencia de b (y $\overline{n}$ es una potencia de b mayor o igual que n_0), $\overline{n} = b\underline{n}$, f es b-uniforme (y $\underline{n} > n/b \geq n_0$), y f es asintóticamente no decreciente ($n \geq \underline{n} > n_0$). Esto demuestra que $t(n) \leq acf(n)$ para *todos* los valores de $n \geq \max(1, bn_0)$, y por tanto $t(n) \in O(f(n))$. La demostración de que $t(n) \in \Omega(f(n))$ es parecida.

3.5 NOTACIÓN ASINTÓTICA CON VARIOS PARÁMETROS

Puede suceder, cuando se analiza un algoritmo, que su tiempo de ejecución dependa simultáneamente de más de un parámetro del ejemplar en cuestión. Esta situación es típica de ciertos algoritmos para problemas de grafos, por ejemplo, en los cuales el tiempo depende tanto del número de nodos como del número de aristas. En tales casos la noción «tamaño del ejemplar» que se ha utilizado hasta el momento, puede perder gran parte de su significado. Por esta razón, se generaliza la notación asintótica de forma natural para funciones de varias variables.

Sea $f: \mathbb{N} \times \mathbb{N} \to \mathbb{R}^{\geq 0}$ una función de parejas de números naturales en los reales no negativos, tal como $f(m, n) = m \log n$. Sea $t: \mathbb{N} \times \mathbb{N} \to \mathbb{R}^{\geq 0}$ otra función de éstas. Diremos que $t(m, n)$ es del orden de $f(m, n)$, lo cual se denota $t(m, n) \in O(f(m, n))$ si $t(m, n)$ está acotada superiormente por un múltiplo positivo de $f(m, n)$ siempre que tanto m como n sean suficientemente grandes. Formalmente, $O(f(m, n))$ se define en la forma

$$\{t: \mathbb{N} \times \mathbb{N} \to \mathbb{R}^{\geq 0} \mid (\exists c \in \mathbb{R}^+)\ (\forall\ m, n \in \mathbb{N})\ [t(m, n) \leq cf(m, n)]\}$$

(No hay necesidad de utilizar dos umbrales distintos para m y n en $\forall\, m, n \in$.) La generalización a más de dos parámetros, de la notación asintótica condicional y de las notaciones Θ y Ω, se hace de forma similar.

La notación asintótica con varios parámetros es parecida a lo que hemos visto hasta el momento, salvo por una diferencia esencial: la regla del umbral ya no es válida. De hecho, el umbral es indispensable en algunas ocasiones. La razón es que aun cuando nunca hay más de un número finito de números no negativos que sean menores que cualquier umbral dado, existe en general un número infinito de parejas $\langle m, n \rangle$ de números no negativos tales que o bien m o bien n está por debajo del umbral; véase el problema 3.32. Por esta razón, $O(f(m, n))$ puede tener sentido aunque $f(m, n)$ sea negativa o no esté definida para un conjunto infinito de puntos, siempre y cuando todos estos puntos se puedan descartar mediante una selección adecuada del umbral.

3.6 OPERACIONES SOBRE NOTACIÓN ASINTÓTICA

Para simplificar algunos cálculos, podemos manipular la notación asintótica empleando operadores aritméticos. Por ejemplo, $O(f(n)) + O(g(n))$ representa el conjunto de operaciones obtenidas sumando punto a punto toda función de $O(f(n))$ a cualquier función de $O(g(n))$. Intuitivamente este conjunto representa el orden del tiempo requerido por un algoritmo compuesto por una primera fase que requiere un tiempo del orden de $f(n)$ seguido por una segunda fase que requiera un tiempo del orden de $g(n)$. Hablando con propiedad, las constantes ocultas que multiplican a $f(n)$ y $g(n)$ pueden muy bien ser distintas, pero esto no tiene importancia porque es sencillo demostrar que $O(f(n)) + O(g(n))$ es idéntico a $O(f(n) + g(n))$. Por la regla del máximo, sabemos que esto también es lo mismo que $O(\max(f(n), g(n)))$ y, si se prefiere, $\max(O(f(n)), O(g(n)))$.

Más formalmente si **op** es un operador binario y si X e Y son conjuntos de funciones de en , en particular conjuntos descritos mediante la notación asintótica, entonces «X **op** Y» denota el conjunto de funciones que se pueden obtener seleccionando una función de X y una función de Y, y «**op**-erando» uno con otro punto a punto. Siguiendo el espíritu de la notación asintótica, solamente exigimos que la función resultante sea el valor operado correcto más allá de un cierto umbral. Formalmente, X **op** Y denota

$$\{t: \times \to \mid (\exists f \in X)\ (\exists g \in Y)\ (\forall\, n \in)\ [t(n) = f(n)\ \mathbf{op}\ g(n)]\}$$

Si g es una función de en , extenderemos la notación escribiendo g **op** X para denotar $\{g\}$ **op** X, el conjunto de funciones que se pueden obtener operando la función g con una función de X. Además, si $a \in$ utilizamos a **op** X para denotar cst_a **op** X, donde cst_a denota la función constante $cst_a(n) = a$ para todo entero n. En otras palabras, a **op** X denota el conjunto de funciones que se pueden obtener operando a con el valor de una función de X. También denotaremos la notación

simétrica X **op** g y X **op** a, y toda esta teoría de operaciones sobre conjuntos se extiende de forma evidente a los operadores que no sean binarios y a las funciones.

Como ejemplo, consideremos el significado de $n^{O(1)}$. Aquí, «n» y «1» denotan la función identidad $Id(n)=n$ y la función $cst_1(n)=1$, respectivamente. Por tanto, una función $t(n)$ pertenece a $n^{O(1)}$ si existe una función $f(n)$ acotada superiormente por una constante c tal que $t(n) = n^{f(n)}$ para todo $n \geq n_0$ para algún n_0. En particular esto implica que $t(n) \leq k + n^k$ para todo $n \geq 0$, siempre y cuando $k \geq c$ y $k \geq t(n)$ para todo $n < n_0$. Por tanto, toda función de $n^{O(1)}$ está acotada superiormente por un polinomio. A la inversa, considérese cualquier polinomio $p(n)$ cuyo primer coeficiente sea positivo y cualquier función $t(n)$ tal que $1 \leq t(n) \leq p(n)$ para todo n suficientemente grande. Sea k el grado de $p(n)$. Es fácil demostrar que $p(n) \leq n^{k+1}$ para tanto, $g(n) \in O(1)$, lo cual demuestra que $t(n) \in n^{O(1)}$. La conclusión es que $n^{O(1)}$ reque $1 \leq t(n) \leq n^{k+1}$, se sigue que $0 \leq g(n) \leq k + 1$ para n suficientemente grande. Por tanto, $g(n) \in O(1)$, lo cual demuestra que $t(n) \in n^{n^{O(1)}}$. La conclusión es que $n^{n^{O(1)}}$ representa el conjunto de todas las funciones acotadas superiormente por algún polinomio, siempre y cuando la función esté acotada inferiormente por la constante uno para todo n suficientemente grande.

3.7 PROBLEMAS

Problema 3.1. Considérese una implementación del algoritmo que requiera un tiempo que esté acotado superiormente por la improbable función

$$t(n) = 3 \text{ segundos} - 18n \text{ milisegundos} + 27n^2 \text{ microsegundos}$$

para resolver un ejemplar de tamaño n. Hallar la función más sencilla posible $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ tal que el algoritmo requiera un tiempo en el orden de $f(n)$.

Problema 3.2. Considérense dos algoritmos A y B que requieren un tiempo en $\Theta(n^2)$ y $\Theta(n^3)$, respectivamente, para resolver el mismo problema. Si los demás recursos, tal como el espacio de almacenamiento y el tiempo de programación no son relevantes, ¿podremos estar seguros de que el algoritmo A siempre será preferible al algoritmo B? Justifique su respuesta.

Problema 3.3. Considérense dos algoritmos A y B que requieran un tiempo en $O(n^2)$ y en $O(n^3)$, respectivamente. ¿Podría existir una implementación del algoritmo B que fuera más eficiente (en términos de tiempo de ejecución) que una implementación del algoritmo A para *todos* los ejemplares? Justifique su respuesta.

Problema 3.4. ¿Qué significa $O(1)$? ¿Qué significa $\Theta(1)$?

Problema 3.5. ¿Cuáles de las siguientes afirmaciones son verdaderas? Demuestre sus respuestas.

1. $n^2 \in O(n^3)$
2. $n^2 \in \Omega(n^3)$
3. $2^n \in \Theta(2^{n+1})$
4. $n! \in \Theta((n+1)!)$

Problema 3.6. Demuestre que si $f(n) \in O(n)$ entonces $[f(n)]^2 \in O(n^2)$.

Problema 3.7. En contraste con el problema 3.6, demostrar que $2^{f(n)} \in O(2^n)$ no se sigue necesariamente de $f(n) \in O(n)$.

Problema 3.8. Considere un algoritmo que requiera un tiempo en el orden de $\Theta(n^{\lg 3})$ para resolver ejemplares de tamaño n. ¿Es correcto decir que requiere un tiempo en el orden de $O(n^{1.58})$? ¿En el orden de $\Omega(n^{1.58})$? ¿En el orden de $\Theta(n^{1.58})$? Justifique sus respuestas. (*Nota:* $\lg 3 \approx 1.58496\ldots$).

Problema 3.9. Demostrar que la notación O es reflexiva: $f(n) \in O(f(n))$ para cualquier función $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$.

Problema 3.10. Demostrar que la notación O es transitiva: se sigue de

$$f(n) \in O(g(n)) \text{ y } g(n) \in O(h(n))$$

que $f(n) \in O(h(n))$ para cualesquiera funciones $f, g, h: \mathbb{N} \to \mathbb{R}^{\geq 0}$.

Problema 3.11. Demostrar que la ordenación de funciones inducida mediante la notación O no es total: proporcionar explícitamente dos funciones $f, g: \mathbb{N} \to \mathbb{R}^{\geq 0}$ tal que $f(n) \notin O(g(n))$ y $g(n) \notin O(f(n))$. Demuestre su respuesta.

Problema 3.12. Demostrar que la notación Ω es reflexiva y transitiva, para cualesquiera funciones $f, g, h: \mathbb{N} \to \mathbb{R}^{\geq 0}$:

1. $f(n) \in \Omega(f(n))$
2. Si $f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(h(n))$ entonces $f(n) \in \Omega(h(n))$.

En lugar de demostrar estas propiedades directamente (¡lo cual sería más sencillo!), utilice la regla de dualidad y los resultados de los problemas 3.9 y 3.10.

Problema 3.13. Tal como hemos explicado, algunos autores dan una definición diferente para la notación Ω. A efectos de este problema, diremos que $t(n) \in \tilde{\Omega}(f(n))$ si existe una constante real positiva d tal que $t(n) \geq df(n)$ para un *número infinito* de valores de n. Formalmente:

$$\tilde{\Omega}(f(n)) = \{t: \mathbb{N} \to \mathbb{R}^{\geq 0} \mid (\exists d \in \mathbb{R}^{+})(\exists^{\infty} n \in \mathbb{N})[t(n) \geq df(n)]\}.$$

Demostrar que esta notación no es transitiva. Específicamente, dar un ejemplo explícito de tres funciones $f, g, h: \mathbb{N} \to \mathbb{R}^{+}$ tales que $f(n) \in \tilde{\Omega}(g(n))$ y $g(n) \in \tilde{\Omega}(h(n))$, y sin embargo $f(n) \notin \tilde{\Omega}(h(n))$.

Problema 3.14. Sea $f(n) = n^2$. Hallar el error en la siguiente «demostración» por inducción matemática de que $f(n) \in O(n)$.

Base: El caso $n = 1$ se satisface de forma trivial puesto que $f(1) = 1 \leq cn$, donde $c = 1$.

Paso de inducción: considérese cualquier $n > 1$. Supóngase por hipótesis de inducción la existencia de una constante positiva c tal que $f(n-1) \leq c(n-1)$:

$$f(n) = n^2 = (n-1)^2 + 2n - 1 = f(n-1) + 2n - 1$$
$$\leq c(n-1) + 2n - 1 = (c+2)n - c - 1 < (c+2)n$$

Entonces hemos mostrado según se pretendía la existencia de una constante $\hat{c} = c + 2$ tal que $f(n) \leq \hat{c}n$. Se sigue, por el principio de inducción matemática, que $f(n)$ está acotada superiormente por una constante n que multiplica a n para todo ≥ 1 y por tanto que $f(n) \in O(n)$ por definición de la notación O.

Problema 3.15. Hallar el error en la siguiente «demostración» de que $O(n) = O(n^2)$. Sea $f(n) = n^2$, $g(n) = n$ y $h(n) = g(n) - f(n)$. Está claro que $h(n) \leq g(n) \leq f(n)$ para todo $n \geq 0$. Por tanto, $f(n) = \max(f(n), h(n))$. Empleando la regla del máximo, concluiríamos que

$$O(g(n)) = O(f(n) + h(n)) =$$
$$O(\max(f(n), h(n))) = O(f(n)).$$

Problema 3.16. Demostrar por inducción matemática que la regla del máximo se puede aplicar a más de dos funciones. Para ser más exactos, sea k un entero y sean $f_1, f_2, \ldots f_k$ funcio-

nes de $\mathbb{N}$ en $\mathbb{R}^{\geq 0}$. Definimos $g(n) = \max(f_1(n), f_2(n), \ldots, f_k(n))$ y $h(n) = f_1(n) + f_2(n) + \ldots + f_k(n)$ para todo $n \geq 0$. Demostrar que $O(g(n)) = O(h(n))$.

Problema 3.17. Hallar el error en la siguiente demostración de que $O(n)=O(n^2)$.

$$O(n) = O(\max(\underbrace{n, n, \ldots, n}_{n \text{ veces}})) = O(\underbrace{n + n + \ldots + n}_{n \text{ veces}}) = O(n^2)$$

en donde la igualdad en la parte central viene de la regla del máximo generalizada que se demostró en el problema 3.16.

Problema 3.18. Demostrar que la notación Θ es reflexiva, simétrica y transitiva: para cualesquiera funciones $f,g,h: \mathbb{N} \to \mathbb{R}^{\geq 0}$:

1. $f(n) \in \Theta(f(n))$
2. Si $f(n) \in \Theta(g(n))$ entonces $g(n) \in \Theta(f(n))$
3. Si $f(n) \in \Theta(g(n))$ y $g(n) \in \Theta(h(n))$ entonces $f(n) \in \Theta(h(n))$.

Problema 3.19. Para cualesquiera funciones $f,g: \mathbb{N} \to \mathbb{R}^{\geq 0}$ demostrar que

$$O(f(n)) = O(g(n)) \text{ si y sólo si } f(n) \in \Theta(g(n))$$
$$\text{si y sólo si } \Theta(f(n)) = \Theta(g(n)).$$

Problema 3.20. Para cualesquiera funciones $f,g: \mathbb{N} \to \mathbb{R}^{\geq 0}$, demostrar que

$$O(f(n)) \subset O(g(n)) \text{ si y sólo si } f(n) \in O(g(n))$$
$$\text{pero } f(n) \notin \Omega(g(n))$$

Recuérdese que «$\subset$» denota la inclusión *estricta* de conjuntos.

Problema 3.21. Para ilustrar la forma en que se puede utilizar una notación asintótica para ordenar la eficiencia de los algoritmos, utilícense las relaciones «$\subset$» y «=» para poner los órdenes de las siguientes funciones en una secuencia, en donde ε es una constante real positiva arbitraria, $0 < \varepsilon < 1$:

$$\log n \quad n^{\varepsilon} \quad n^{1-\varepsilon} \quad (1+\varepsilon)^n \quad n^2 / \log n \quad (n^2 - n + 1)!$$

No utilice el símbolo «$\subseteq$». Demuestre sus respuestas.

Problema 3.22. Repita el problema 3.21 pero esta vez con las funciones

$$n! \quad (n+1)! \quad 2^n \quad 2^{n+1} \quad 2^{2n} \quad 2^n \quad n^n \quad n^{n-n} \quad n^{\log n}$$

Problema 3.23. Vimos al final de la Sección 3.3 que $\sum_{i=1}^{n} i^k \in \Omega(n^{k+1})$ para cualquier entero prefijado $k \geq 0$ porque $\sum_{i=1}^{n} i^k \geq n^{k+1}/2^{k+1}$ para todo n. Utilice la idea que subyace a la figura 1.8 de la Sección 1.7.2 para derivar una constante más ajustada para esta desigualdad: busque una constante d(dependiente de k) mucho más grande que $1/2^{k+1}$ y tal que $\sum_{i=1}^{n} i^k \geq dn^{k+1}$ sea válido para todo n. *No* utilice la proposición 1.7.16.

Problema 3.24. Demuestre que $\log(n!) \in \Theta(n \log n)$. *No* utilice la fórmula de Stirling.

Sugerencia: Imite la demostración de que $\sum_{i=1}^{n} i^k \in \Omega(n^{k+1})$ que se daba al final de la Sección 3.3. Resista la tentación de mejorar su razonamiento siguiendo las líneas del problema 3.23.

Problema 3.25. Recuerde que una función $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ es asintóticamente no decreciente si existe un umbral entero n_0 tal que $f(n) \leq f(n+1)$ para todo $n \geq n_0$. Demuestre por inducción matemática que esto implica que $f(n) \leq f(m)$ siempre que $m \geq n \geq n_0$.

Problema 3.26. Proporcionar una función $t: \mathbb{N} \to \mathbb{R}^{\geq 0}$ tal que $t(n) \notin \Theta(f(n))$ siempre que $f(n)$ sea una función asintóticamente no decreciente. Dé un ejemplo *natural* de algoritmo cuyo tiempo de ejecución esté en el orden de $\Theta(t(n))$. Quizá pudiera considerar algoritmos numéricos en los cuales n sea el valor del caso en lugar de ser el valor del tamaño.

Problema 3.27. Considere cualquier función b-uniforme $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$. Sean c y n_0 cons-

tantes tales que $f(bn) \leq cf(n)$ para todo $n \geq n_0$. Considere cualquier entero positivo i. Demuestre por inducción matemática que $f(b^i n) \leq c^i f(n)$ para todo $n \geq n_0$.

Problema 3.28. Considere la función t: $\mathbb{N} \to \mathbb{R}^{\geq 0}$ definida recursivamente mediante

$$t(n) = \begin{cases} a & \text{si } n = 1 \\ 4t(\lceil n/2 \rceil) + bn & \text{en caso contrario} \end{cases}$$

en donde a y b son constantes positivas arbitrarias (ésta era la ecuación 3.2 de la Sección 3.4). Demuestre por inducción matemática que $t(n)$ es eventualmente no decreciente.

Problema 3.29. Complete la demostración de la regla de uniformidad (final de la Sección 3.4) proporcionando detalles acerca de la demostración de que $t(n) \in \Omega(f(n))$ siempre que $f(n)$ sea una función uniforme y $t(n)$ sea una función eventualmente no decreciente tal que $t(n) \in \Theta(f(n) \mid n$ es una potencia de $b)$.

Obsérvese que la regla de uniformidad se aplica igualmente bien si sustituimos Θ por O o por Ω en esta afirmación. Esto se debe a que la demostración de que $t(n) \in \Omega(f(n))$ no utiliza que $t(n) \in O(f(n) \mid n$ es una potencia de $b)$, tal como la demostración que se daba en la Sección 3.4 de que $t(n) \in O(f(n))$ no utilizaba que $t(n) \in \Omega(f(n) \mid n$ es una potencia de $b)$.

Sugerencia: Esto es muy parecido a la demostración de que $t(n) \in O(f(n))$ bajo las mismas condiciones. Utilice el hecho consistente en que si $f(bm) \leq cf(m)$ entonces $f(m) \geq c^{-1} f(bm)$.

Problema 3.30. Mostrar mediante ejemplos explícitos que todas las condiciones previas para aplicar la regla de la uniformidad son necesarias. Específicamente nos interesan las funciones f, t: $\mathbb{N} \to \mathbb{R}^{\geq 0}$ tales que $t(n) \in \Theta(f(n) \mid n$ es una potencia de $b)$ para algún entero $b \geq 2$, y sin embargo, $t(n) \notin \Theta(f(n))$. Hay que dar tres pares de funciones como éstas, sometidas además a las restricciones siguientes:

1. $f(n)$ es uniforme, pero $f(n)$ no es asintóticamente no decreciente.
2. $f(n)$ y $t(n)$ son ambas asintóticamente no decrecientes, pero $f(bn) \notin O(f(n))$.
3. $f(bn) \in O(f(n))$ y $t(n)$ es asintóticamente no decreciente, pero $f(n)$ no es asintóticamente no decreciente.

Problema 3.31. Mostrar mediante un ejemplo explícito que ser eventualmente no decreciente y estar acotada superiormente por un polinomio no es garantía de uniformidad. En otras palabras, dar dos funciones f, p: $\mathbb{N} \to \mathbb{R}^{\geq 0}$ tales que $f(n)$ sea eventualmente no decreciente, $p(n)$ sea un polinomio y $f(n) \leq p(n)$ para todo n, y sin embargo $f(n)$ no sea suave.

Problema 3.32. Mostrar que la regla del umbral no es aplicable a funciones de varios parámetros. Específicamente, dar un ejemplo explícito de dos funciones f, g: $\mathbb{N} \times \mathbb{N} \to \mathbb{R}^+$ tales que $f(m, n) \in O(g(m, n))$, y sin embargo no existe una constante c tal que $f(m, n) \leq cg(m, n)$ para todo $m, n \in \mathbb{N}$.

Problema 3.33. Considérense dos funciones cualesquiera $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. Demostrar que

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\text{máx}(f(n), g(n))) = \text{máx}(O(f(n)), O(g(n))).$$

Nota: Ya sabemos que $O(f(n) - g(n)) = O(\text{máx}(f(n), g(n)))$ por la regla del máximo.

Problema 3.34. Demostrar que $\Theta(n-1) + \Theta(n) = \Theta(n)$. ¿Se sigue que $\Theta(n) = \Theta(n) - \Theta(n-1)$?

Justifique su respuesta.

Problema 3.35. Hallar una función $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ que esté acotada superiormente por algún polinomio, y sin embargo tal que $f(n) \notin n^{O(1)}$.

3.8 REFERENCIAS Y TEXTOS MÁS AVANZADOS

La notación asintótica O ya existe desde hace algún tiempo en matemáticas: véase Bachmann (1894) y de Bruijn (1961). Sin embargo, la notación Θ y Ω es más reciente: fue inventada para analizar algoritmos y para la teoría de complejidad computacional. Knuth (1976) ofrece una descripción de la historia de la notación asintótica, y propone un estándar para ella. La notación condicional asintótica y la noción de uniformidad, ajuste o suavización han sido introducidas por Brassard (1985), que también sugirió que las «igualdades de una sola dirección» debían abandonarse, pasando a una notación basada en conjuntos.

Capítulo 4

Análisis de algoritmos

4.1 INTRODUCCIÓN

El objetivo principal de este libro es enseñarle a diseñar sus propios algoritmos eficientes. Sin embargo, cuando uno se enfrenta con varios algoritmos distintos para resolver el mismo problema, es preciso decidir cuál de ellos es el más adecuado para la aplicación considerada. Una herramienta esencial para este propósito es el *análisis de algoritmos*. Sólo después de haber determinado la eficiencia de los distintos algoritmos será posible tomar una decisión bien informada. Pero no hay una fórmula mágica para analizar la eficiencia de los algoritmos. En su mayor parte, es una cuestión de juicio, intuición y experiencia. Sin embargo, existen algunas técnicas básicas que suelen resultar útiles, tales como saber la forma de enfrentarse a estructuras de control y a ecuaciones de recurrencia. Este capítulo abarca las técnicas de uso más frecuente, y las ilustra con ejemplos. Se hallarán más ejemplos a lo largo de todo el libro.

4.2 ANÁLISIS DE LAS ESTRUCTURAS DE CONTROL

El análisis de los algoritmos suele efectuarse desde dentro hacia afuera. En primer lugar, se determina el tiempo requerido por las instrucciones individuales (este tiempo suele estar acotado por una constante); después se combinan estos tiempos de acuerdo con las estructuras de control que enlazan las instrucciones del programa. Algunas estructuras de control, como la composición secuencial—poner una instrucción tras otra—son fáciles de analizar, mientras que los bucles **mientras**[1] son más sutiles. En esta sección, ofreceremos unos principios generales que resultan útiles en aquellos análisis relacionados con las estructuras de control de uso más frecuente, así como ejemplos de la aplicación de estos principios.

4.2.1 Secuencias

Sean P_1 y P_2 dos fragmentos de un algoritmo. Pueden ser instrucciones individuales o bien subalgoritmos complicados. Sean t_1 y t_2 los tiempos requeridos por P_1 y P_2 respectivamente. Estos tiempos pueden depender de distintos parámetros, tales como el tamaño del caso. La **regla de la composición secuencial** dice que el tiempo necesario para calcular "P_1; P_2", esto es, primero P_1 y después P_2, es simplemente $t_1 + t_2$. Por la regla del máximo este tiempo está en $\Theta(\text{máx}\,(t_1, t_2))$.

A pesar de su sencillez, la aplicación de esta regla puede ser menos fácil de lo que parece en principio. Por ejemplo, puede ser que uno de los parámetros que controlan t_2 dependa del resultado del cálculo efectuado por P_1. Por tanto, el análisis de "P_1; P_2" no siempre se puede efectuar considerando P_1 y P_2 independientemente.

4.2.2 Bucles "para" (desde)

Los bucles[2] (lazos) **para (desde)**[3] son los bucles más fáciles de analizar. Considérese el bucle siguiente.

para $i \leftarrow 1$ **hasta** m **hacer** $P(i)$

Tanto aquí como en el resto del libro adoptamos el convenio de que $m = 0$ no es un error; significa simplemente que la instrucción controlada $P(i)$ no se ejecuta ni una sola vez. Supongamos que este bucle es parte de un algoritmo más extenso, que trabaja en un ejemplar de tamaño n. (Tenga cuidado para no confundir n con m.) El caso más sencillo es aquel en el cual el tiempo requerido por $P(i)$ no depende realmente de i, aun cuando pudiera depender del tamaño del ejemplar o, más generalmente, del ejemplar en sí. Supongamos que t denota el tiempo requerido para calcular $P(i)$. En este caso, el análisis evidente del bucle es que $P(i)$ se efectúa m veces, cada una de las cuales tiene un coste t, y por tanto el tiempo total requerido por el bucle es simplemente $t' = mt$. Aun cuando este enfoque suele ser apropiado, existe un posible problema: no hemos tenido en cuenta el tiempo necesario para el *control del bucle*. Después de todo, nuestro bucle **for** es una abreviatura de algo parecido al siguiente bucle **mientras**:

$i \leftarrow 1$
mientras $i \leq m$ **hacer**
    $P(i)$
    $i \leftarrow i + 1$

En casi todas las situaciones es razonable asignar un coste unitario a la comprobación $i \leq m$, a las instrucciones $i \leftarrow 1$ e $i \leftarrow i + 1$, y a las instrucciones de salto (**ir a**) implícitas en el bucle **mientras**. Sea c una cota superior del tiempo requerido por cada una de estas operaciones. Entonces el tiempo t' requerido por el bucle está acotado superiormente por

$$
\begin{aligned}
t &\le c && \text{para } i \leftarrow 1 \\
&\quad + (m+1)c && \text{para las comprobaciones de } i \le m \\
&\quad + mt && \text{para las ejecuciones de } P(i) \\
&\quad + mc && \text{para las ejecuciones de } i \leftarrow i+1 \\
&\quad + mc && \text{para las instrucciones de salto} \\
&\le (t+3c)m + 2c
\end{aligned}
$$

Además este tiempo está, claramente, acotado inferiormente por mt. Si c es despreciable en comparación con t, entonces nuestra estimación anterior de t como aproximadamente igual a mt queda justificada, salvo por un caso crucial: $t \approx mt$ es completamente incorrecto cuando $m = 0$ (¡resulta incluso peor cuando m es negativo!). Veremos en la Sección 4.3 que despreciar el tiempo necesario para el control del bucle puede dar lugar a graves errores en tales circunstancias.

Resista la tentación de decir que el tiempo requerido por el bucle está en $\Theta(mt)$ con el pretexto de la notación Θ sólo tiene efecto más allá de algún umbral tal como $m \ge 1$. El problema de este argumento es que si estamos realmente analizando todo un algoritmo y no simplemente el bucle **para**, entonces el umbral implícito en la notación Θ concierne a n, el tamaño del caso, y no a m, el número de veces que pasamos por el bucle, y $m = 0$ podría suceder para valores arbitrariamente grandes de n. Por otra parte, siempre y cuando t esté acotado *inferiormente* por alguna constante (lo cual siempre es cierto en la práctica), y siempre y cuando exista un umbral n_0 tal que $m \ge 1$ siempre que $n \ge n_0$, el problema 4.3 pide demostrar que t está ciertamente en $\Theta(mt)$ cuando t, m y t se consideran como funciones de n.

El análisis de bucles **para** es más interesante cuando el tiempo $t(i)$ requerido por $P(i)$ varía como función de i. (En general, el tiempo requerido por $P(i)$ podría depender no sólo de i sino también del tamaño del caso, n, o incluso del caso en sí.) Si despreciamos el tiempo requerido por el bucle de control, lo cual suele ser correcto siempre que $m \ge 1$, entonces ese mismo bucle **para** requiere un tiempo que no está dado por una multiplicación, sino por una suma: se trata de $\sum_{i=1}^{m} t(i)$. Las técnicas de la Sección 1.7.2 suelen resultar útiles para traducir estas sumas a la notación asintótica, más sencilla.

Ilustraremos el análisis de bucles **para** con un sencillo algoritmo para calcular la sucesión de Fibonacci que evaluábamos empíricamente en la Sección 2.7.5. Repetimos a continuación el algoritmo:

```
función Fibiter(n)
    i ← 1; j ← 0
    para k ← 1 hasta n hacer j ← i + j
                             i ← j − i
    devolver j
```

Si contamos todas las operaciones aritméticas como de coste unitario, las instrucciones que están dentro del bucle **para** requieren un tiempo constante. Su-

pongamos que el tiempo requerido por estas instrucciones está acotado superiormente por alguna constante c. Sin tener en cuenta el control del bucle, el tiempo requerido por el bucle **para** está acotado superiormente por n veces esta constante: nc. Dado que las instrucciones anteriores y posteriores al bucle requieren un tiempo despreciable, concluimos que el algoritmo requiere un tiempo que está en $O(n)$. Un razonamiento similar indica que este tiempo también está en $\Omega(n)$, puesto que está en $\Theta(n)$.

Sin embargo, vimos en la Sección 2.5 que no es razonable tomar como de coste unitario las sumas implicadas en el cómputo de la sucesión de Fibonacci, a no ser que n sea muy pequeña. Por tanto, deberíamos tener en cuenta el hecho consistente en que una instrucción tan sencilla como "$j \leftarrow i + j$" se vuelve más costosa a cada pasada por el bucle. Es fácil programar adiciones y sustracciones de enteros muy grandes de tal modo que el tiempo requerido para sumar o restar dos enteros sea del orden exacto del número de cifras que tenga el mayor de los operandos. Para determinar el tiempo requerido por la k-ésima pasada por el bucle, necesitamos conocer la longitud de los enteros implicados. El problema 4.4 pide demostrar por inducción matemática que los valores de i y de j al final de la k-ésima iteración son f_{k-1} y f_k respectivamente. Ésta es precisamente la razón por la cual funciona el algoritmo: devuelve el valor de j al final de la n-ésima iteración, que es precisamente f_n según deseábamos. Además, se vio en la Sección 2.7.5 que la fórmula de Moivre nos dice que el tamaño de f_k está en $\Theta(k)$. Por tanto, la k-ésima necesita un tiempo en $\Theta(k-1)+\Theta(k)$ véase el problema 3.34. Sea c una constante tal que este tiempo está acotado superiormente por ck para todo $k \geq 1$. Si despreciamos el tiempo requerido por el control del bucle y por las instrucciones que preceden al bucle (y por las que lo siguen), concluiremos que el tiempo requerido por el algoritmo está acotado superiormente por

$$\sum_{k=1}^{n} ck = c\sum_{k=1}^{n} k = c\frac{n(n+1)}{2} \in O(n^2)$$

Un razonamiento similar indica que este tiempo se encuentra en $\Omega(n^2)$, y que por tanto está en $\Theta(n^2)$. Por tanto, contar o no como de coste unitario las operaciones aritméticas supone una diferencia crucial en el análisis de Fibiter.

El análisis de bucles **para** que empiezan en un valor que no sea 1, o que avanzan con pasos mayores, debería resultar evidente a partir de lo dicho. Considérese el siguiente ejemplo de bucle:

para $i \leftarrow 5$ **hasta** m **paso** 2 **hacer** $P(i)$

Aquí, $P(i)$ se ejecuta $((m-5) \div 2) + 1$ veces siempre que $m \geq 3$. (Para que un bucle **para** tenga sentido, el punto final siempre debería ser tan grande como el punto inicial *menos* el paso.)

4.2.3 Llamadas recursivas

El análisis de algoritmos recursivos suele ser sencillo, al menos hasta cierto punto. Una inspección sencilla del algoritmo suele dar lugar a una ecuación de

recurrencia que "remeda" (imita) el flujo de control dentro del algoritmo. Una vez que se ha obtenido la ecuación de recurrencia, se pueden aplicar las técnicas generales descritas en la Sección 4.7 para transformar la ecuación en la ecuación asintótica no recursiva, que es más sencilla.

Como ejemplo, considérese de nuevo el problema consistente en calcular la sucesión de Fibonacci, pero esta vez con el algoritmo recursivo Fibrec, que comparábamos con Fibiter en la Sección 2.7.5.

```
función Fibrec(n)
    si n < 2 entonces devolver n
    sino devolver Fibrec(n - 1) + Fibrec(n - 2)
```

Sea $T(n)$ el tiempo requerido por una llamada a *Fibrec*(n). Si $n < 2$, el algoritmo devuelve simplemente n, lo cual requiere algún tiempo constante a. En caso contrario, la mayor parte del trabajo se invierte en las dos llamadas recursivas, que requieren un tiempo $T(n-1)$ y $T(n-2)$ respectivamente. Además, hay que efectuar una suma de f_{n-1} y f_{n-2} (que son los valores proporcionados por las llamadas recursivas), y también hay que efectuar el control de la recursividad y la comprobación "**si** $n < 2$". Sea $h(n)$ el trabajo implicado en esta suma y en este control, esto es, el tiempo requerido por una llamada a *Fibrec*(n) ignorando los tiempos invertidos dentro de las dos llamadas recursivas. Por definición de $T(n)$ y de $h(n)$, obtenemos la siguiente recurrencia:

$$T(n) = \begin{cases} a & \text{si } n = 0 \text{ ó } n = 1 \\ T(n-1) + T(n-2) + h(n) & \text{en caso contrario} \end{cases} \qquad (4.1)$$

Si contamos las sumas con coste unitario, $h(n)$ está acotado por una constante, y la ecuación de recurrencia para $T(n)$ es muy similar a la que ya hemos encontrado para g(n) en la Sección 1.6.4. La inducción constructiva se aplica igualmente bien para alcanzar la misma conclusión: $T(n) \in O(f_n)$. Sin embargo, es más sencillo en este caso aplicar la técnica de la Sección 4.7 para resolver la recurrencia 4.1. Un razonamiento similar muestra que $T(n) \in \Omega(f_n)$, y por tanto $T(n) \in \Theta(f_n)$. Empleando la fórmula de Moivre, concluimos que Fibrec(n) requiere un tiempo exponencial en n. Es decir *doblemente* exponencial en el tamaño del caso, puesto que el *valor* de n es exponencial en el *tamaño* de n.

Si no se cuentan las adiciones con un coste unitario, $h(n)$ ya no queda acotado por una constante. En lugar de ocurrir esto, $h(n)$ está dominado por el tiempo requerido para la adición de f_{n-1} y f_{n-2} para n suficientemente grande. Ya hemos visto que esta adición requiere un tiempo que es del orden exacto de n. Por tanto $h(n) \in \Theta(n)$. Las técnicas de la Sección 4.7 son aplicables una vez más para resolver la recurrencia 4.1. Sorprendentemente, el resultado es el mismo independientemente de si $h(n)$ es constante o lineal: sigue sucediendo que $T(n) \in \Theta(f_n)$. En conclusión, *Fibrec*(n) requiere un tiempo exponencial en n (¡tanto si se cuentan las adiciones con coste unitario como si no!). La única diferencia es la constante multiplicativa oculta en la notación Θ.

4.2.4 Bucles "mientras" (while) y "repetir" (repeat)

Los bucles **mientras (while)** y **repetir (repeat)** suelen ser más difíciles de analizar que los bucles **para (for)**, porque no existe una forma evidente *a priori* de saber cuántas veces tendremos que pasar por el bucle. La técnica estándar para analizar estos bucles es hallar una función de las variables implicadas cuyo valor se decremente en cada pasada. Para concluir que el bucle terminará por acabar, basta con saber que este valor debe ser un entero positivo. (No se puede seguir decrementando indefinidamente un mínimo natural.) Para determinar el número de veces que se repite el bucle, sin embargo, necesitamos conocer mejor la forma en que disminuye el valor de esta función. Una aproximación alternativa al análisis del bucle **mientras** consiste en tratarlo como un algoritmo recursivo. Ilustraremos ambas técnicas con el mismo ejemplo. El análisis de bucles **repetir** se efectúa de manera similar; no daremos ejemplos de ellos en esta sección.

Estudiaremos con detalle la *búsqueda binaria* en la Sección 7.3. Sin embargo, la utilizamos ahora porque ilustra perfectamente el análisis de los bucles **mientras**. El objetivo de la búsqueda binaria es hallar un elemento x de un vector $T[1..n]$ que está ordenado de modo no decreciente. Supongamos por sencillez que está garantizado que x aparece al menos una vez en T. (El caso general se trata en la Sección 7.3.) Se nos pide buscar un entero i tal que $1 \leq i \leq n$ y $T[i] = x$. La idea básica que subyace a la búsqueda binaria es comparar x con el elemento y que está en la posición media de T. La búsqueda concluye si $x = y$; se puede limitar a la mitad superior de la matriz si $x > y$; en caso contrario resulta suficiente examinar la mitad inferior. Obtenemos el siguiente algoritmo (en la Sección 7.3 se da un algoritmo un poco mejor; véase el problema 7.11):

```
función Búsqueda binaria(T[1..n], x)
    { Este algoritmo supone que x se encuentra en T}
    i ← 1; j ← n
    mientras i < j hacer
        { T[i] ≤ x ≤ T[j] }
        k ← (i + j) ÷ 2
        caso_de x < T[k]: j ← k – 1
                x = T[k]: i, j ← k {devolver k}
                x > T[k]: i ← k+1
    devolver i
```

Recuerde que para analizar el tiempo de ejecución de un bucle **mientras** hay que hallar una función de las variables nombradas en el bucle cuyo valor irá decreciendo en cada pasada por el bucle. En este caso, resulta natural considerar $j - i + 1$, expresión que llamaremos d. Por tanto d representa el número de elementos restantes de T que hay que examinar. Inicialmente, $d = n$. El bucle concluye cuando $i \geq j$, lo cual es equivalente a $d \leq 1$. (De hecho, d nunca puede ser menor que 1, pero no utilizaremos esto.) Cada vez que se pasa por el bucle, hay tres posibilidades: o bien se da a j el valor $k - 1$, o bien se da a i el valor $k + 1$, o

bien tanto i como j reciben el valor k. Supongamos que d y $\hat{d}$ representan respectivamente los valores de $j - i + 1$ antes y después de la iteración que estamos considerando. Utilizaremos i, j, $\hat{\imath}$ y $\hat{\jmath}$ de forma similar. Si $x < T[k]$, entonces se ejecuta la instrucción "$j \leftarrow k - 1$" y por tanto $\hat{\imath} = i$ y $\hat{\jmath} = [(i + j) \div 2] - 1$. Por consiguiente,

$$\hat{d} = \hat{\jmath} - \hat{\imath} + 1 = (i + j) \div 2 - i \leq (i + j) \ / \ 2 - i < (j - i + 1)/2 = d/2$$

De manera similar, si $x > T[k]$ entonces se ejecuta la instrucción "$i \leftarrow k + 1$" y por tanto

$$\hat{\imath} = [(i + j) \div 2] + 1 \text{ y } \hat{\jmath} = j$$

Por tanto,

$$\hat{d} = \hat{\jmath} - \hat{\imath} + 1 = j - (i + j) \div 2 \leq j - (i + j - 1)/2 = (j - i + 1)/2 = d/2$$

Finalmente, si $x = T[k]$, entonces i y j reciben el mismo valor y por tanto $\hat{d} = 1$; pero d era al menos 2 puesto que en caso contrario no habríamos vuelto a entrar en el bucle. Concluimos entonces que $\hat{d} \leq d/2$ suceda lo que suceda, lo cual significa que el valor de d se divide cuando menos por dos en cada pasada por el bucle. Dado que nos detenemos cuando $d \leq 1$, el proceso tiene que detenerse eventualmente, pero ¿cuánto tiempo tarda?

Para determinar una cota superior del tiempo de ejecución de la búsqueda binaria, supongamos que d_ℓ denota el valor de $j - i + 1$ al final de la ℓ-ésima vuelta para $\ell \geq 1$, y sea $d_0 = n$. Puesto que $d_{\ell-1}$ es el valor de $j - i + 1$ *antes* de comenzar la ℓ-ésima iteración, hemos demostrado que $d_\ell \leq d_{\ell-1} / 2$ para todo $\ell \geq 1$. Se sigue inmediatamente por inducción matemática que $d_\ell \leq n \ / \ 2^\ell$. Pero el bucle concluye cuando $d \leq 1$, lo cual sucede como mucho cuando $\ell = \lceil \lg n \rceil$. Concluimos que se entra en el bucle, como mucho, $\lceil \lg n \rceil$ veces. Dado que cada pasada por el bucle requiere un tiempo constante, la búsqueda binaria requiere un tiempo que está en $O(\log n)$. Un razonamiento similar produce una cota inferior correspondiente de $\Omega(\log n)$ en el caso peor, y por tanto la búsqueda binaria requiere un tiempo que está en $\Theta(\log n)$. Esto es cierto aun cuando nuestro algoritmo pueda funcionar mucho más rápidamente en el caso mejor en el cual x está situado precisamente en el punto medio del vector; véase el problema 7.11.

El enfoque alternativo para analizar el tiempo de ejecución de la búsqueda binaria comienza de forma muy similar. La idea es pensar en el bucle **mientras** como si estuviera implementado de forma recursiva, en lugar de ser iterativo. Cada vez que se pasa por el bucle, se reduce el rango de posiciones posibles para x dentro de la matriz. Supongamos que $t(d)$ denota el tiempo máximo necesario para terminar el bucle **mientras** cuando $j - i + d \leq d$, esto es, cuando existen como máximo d elementos que todavía hay que considerar. Ya hemos visto que el valor de $j - i + 1$ se divide al menos por dos en cada pasada por el bucle. En términos recursivos, esto significa que $t(d)$ es como máximo el tiempo constante b necesario para pasar una vez por el bucle, más el tiempo $t(d \div 2)$ que basta para finalizar el bucle desde ese punto. Puesto que se requiere un tiempo constante c para determinar que el bucle ha terminado cuando $d = 1$, obtenemos la recurrencia siguiente:

$$t(d) \leq \begin{cases} c & \text{si } d = 1 \\ b + t(d \div 2) & \text{en caso contrario} \end{cases} \qquad (4.2)$$

Las técnicas de la Sección 4.7 se aplican fácilmente para concluir que $t(n) \in O(\log n)$.

4.3 USO DE UN BARÓMETRO

El análisis de muchos algoritmos se simplifica de forma significativa cuando es posible aislar una instrucción —o una comprobación— como *barómetro*. Una "instrucción barómetro" es aquella que se ejecuta por lo menos con tanta frecuencia como cualquier otra instrucción del algoritmo. (No pasa nada si algunas instrucciones se ejecutan como mucho un número constante de veces más que el barómetro, puesto que su contribución quedará absorbida en la notación asintótica.) Siempre que el tiempo requerido por cada instrucción esté acotado por una constante, el tiempo requerido por el algoritmo completo es del orden exacto del número de veces que se ejecuta la instrucción barómetro.

Este enfoque resulta útil porque nos permite despreciar los tiempos exactos que requieren las distintas instrucciones. En particular, evita la necesidad de tener que introducir constantes tales como las que acotan el tiempo requerido por las distintas operaciones elementales, que carecen de significado puesto que dependen de la implementación, y se descartan cuando se expresa el resultado final en términos de la notación asintótica. Por ejemplo, considérese el análisis de *Fibiter* en la Sección 4.2.2 si contamos todas las operaciones aritméticas como de coste unitario. Vimos que el algoritmo requiere un tiempo acotado superior por cn para alguna constante c carente de significado, y por tanto requiere un tiempo que está en $\Theta(n)$. Habría sido más sencillo decir que la instrucción $j \leftarrow i + j$ se puede tomar como barómetro, que esta instrucción es ejecutada evidentemente un número de veces igual a n, y que por tanto el algoritmo requiere un tiempo que está en $\Theta(n)$. La ordenación por selección proporcionará un ejemplo más convincente de cómo utilizar las instrucciones barómetro en la sección siguiente.

Cuando un algoritmo contiene varios bucles anidados, toda instrucción del bucle más interno puede utilizarse en general como barómetro. Sin embargo, hay que hacer esto con cuidado, porque hay casos en los cuales es preciso tener en consideración el control implícito del bucle. Esto sucede, típicamente, cuando algunos de los bucles se ejecutan cero veces, porque estos bucles requieren en realidad un tiempo aun cuando no realizen ejecuciones de la instrucción barómetro. Si esta situación se repite con demasiada frecuencia, el número de veces que se ejecuta la instrucción barómetro puede verse dominado por el número de veces que se entra en bucles vacíos, y por tanto habrá sido un error considerarla como barómetro. Considérese por ejemplo la clasificación por casillas (Sección 2.7.2). Aquí generalizamos el algoritmo para admitir el caso en el cual

los elementos que hay que ordenar son enteros cuyo valor está comprendido entre 1 y s, en lugar de estar entre 1 y 10.000. Recuerde que $T[1..n]$ es el vector que hay que ordenar, y que $U[1..s]$ es un vector construido de tal manera que $U[k]$ da el número de veces que el entero k aparece en T. La fase final del algoritmo reconstruye T por orden no decreciente, a partir de la información disponible en U en la forma siguiente:

```
i ← 0
para k ← 1 hasta s hacer
    mientras U[k] ≠ 0 hacer
        i ← i + 1
        T[i] ← k
        U[k] ← U[k] − 1
```

Para analizar el tiempo requerido por este proceso, utilizamos "$U[k]$" para denotar el valor almacenado *originalmente* en $U[k]$, puesto que todos estos valores se ponen a cero durante el proceso. Resulta tentador seleccionar cualquiera de las instrucciones del bucle interno como barómetro. Para cada valor de k, estas instrucciones se ejecutan $U[k]$ veces. El número total de veces que se ejecutan es por tanto $\Sigma_{k=1}^{n} U[k]$. Pero esta suma es igual a n, el número de enteros que hay que ordenar, puesto que la suma de los números de veces que aparece cada elemento da el número total de elementos. Si fuera verdad que estas instrucciones pueden servir como barómetro, concluiríamos que este proceso requiere un tiempo que está en orden exacto de n. Basta un ejemplo sencillo para convencernos de que esto no es necesariamente cierto. Supongamos que $U[k] = 1$ cuando k es un cuadrado perfecto, y que $U[k] = 0$ en caso contrario. Esta situación se correspondería con ordenar un vector T que contuviese exactamente una vez números que fueran cuadrados perfectos entre 1 y n^2, empleando $s = n^2$ casillas. En este caso, el problema requiere claramente un tiempo que está en $\Omega(n^2)$ puesto que el bucle exterior se ejecuta s veces. Por tanto, *no* puede suceder que el tiempo requerido esté en $\Theta(n)$. Esto demuestra que la selección de instrucciones en el bucle interno como barómetro era incorrecta. Surge el problema porque sólo podemos despreciar el tiempo invertido iniciando y controlando un bucle cuando estemos seguros de que incluimos alguna instrucción cuando el bucle se ejecute cero veces.

El análisis correcto y detallado del proceso es como sigue. Sea a el tiempo necesario para la comprobación $U[k] \neq 0$ en cada pasada por el bucle interno, y sea b el tiempo requerido por una ejecución de las instrucciones del bucle interno, incluyendo la operación implícita de la comprobación que se encuentra al principio del bucle. Para ejecutar el bucle interno completamente para un valor dado de k se necesita un tiempo $t_k = (1 + U[k])a + U[k]b$, entonces añadiríamos 1 a $U[k]$ antes de multiplicarlo por a con objeto de tener en cuenta el hecho de que la comprobación se efectúa una vez en cada pasada por el bucle y *otra* vez más para determinar que el bucle se ha completado. Lo crucial es que este tiempo no es cero aun cuando $U[k] = 0$. El proceso completo requiere un tiempo $c + \Sigma_{k=1}^{s} (d + t_k)\, c$ en donde c y d son nuevas constantes para tener en cuenta el tiempo necesario para ini-

ciar y controlar el bucle exterior, respectivamente. Cuando se simplifica, esta expresión da lugar a $c + (a + d)s + (a + b)n$. Concluimos que el proceso requiere un tiempo que está en $\Theta(n + s)$. Por tanto el tiempo depende de dos parámetros independientes n y s; no se puede expresar como una función de solo uno de ellos. Resulta fácil ver que la fase de inicialización de la ordenación por casillas (Sección 2.7.2) también requiere un tiempo que está en $\Theta(n + s)$, a no ser que se emplee una *iniciación virtual* —véase el final de la Sección 5.1— en cuyo caso basta para esta fase un tiempo en $\Theta(n)$. En todo caso, esta técnica de ordenación requiere un tiempo en $\Theta(n + s)$ en total para ordenar n enteros entre 1 y s. Si se prefiere, se puede invocar la regla del máximo para afirmar que este tiempo está en $\Theta(\text{máx}(n, s))$. Por tanto, la ordenación por casillas merece la pena, pero sólo cuando s sea suficientemente pequeño en comparación con n. Por ejemplo, si estamos interesados en el tiempo requerido como función únicamente del número de elementos que hay que ordenar, esta técnica lo consigue en un asombroso tiempo lineal si $s \in O(n)$, pero este tiempo se convierte en cuadrático cuando $s \in \Theta(n^2)$.

A pesar de lo anterior, el uso del barómetro es adecuado para analizar la ordenación mediante casillas. Nuestro problema era que no seleccionamos el barómetro adecuado. En lugar de las instrucciones que están *dentro* del bucle interno, deberíamos haber utilizado la *comprobación* del bucle interno, "$U[k] \neq 0$", como barómetro. De hecho, ninguna instrucción del proceso se ejecuta más veces que esta comprobación, propiedad que caracteriza exactamente a un barómetro. Es fácil demostrar que esta comprobación se efectúa exactamente $n + s$ veces, y que por tanto la conclusión correcta acerca del tiempo de ejecución del proceso se sigue inmediatamente sin necesidad de introducir constantes carentes de significado.

En conclusión, el uso de un barómetro es una herramienta útil para simplificar el análisis de muchos algoritmos, pero esta técnica deberá utilizarse con cuidado.

4.4 EJEMPLOS ADICIONALES

En esta sección estudiaremos varios ejemplos adicionales de análisis de algoritmos que implican bucles (lazos), recursividad y el uso de barómetros.

Ordenación por selección

La ordenación por selección, que ya vimos en la Sección 2.4, ofrece un buen ejemplo del análisis de bucles *anidados*.

```
procedimiento selección (T[1..n])
    para i ← 1 hasta n -1 hacer
        minj ← i; minx ← T[i]
        para j ← i + 1 hasta n hacer
            si T[j] < minx entonces minj ← j
                                    minx ← T[j]
        T[minj] ← T[i]
        T[i] ← minx
```

Aunque el tiempo invertido por cada pasada del bucle interno no es constante —se tarda más cuando $T[j]<minx$— este tiempo está acotado superiormente por alguna constante c (que tiene en cuenta el control del bucle). Para cada valor de i, las instrucciones del bucle interno se ejecutan $n - (i + 1) + 1 = n - i$ veces, y por tanto el tiempo invertido por el bucle interno es $t(i) \leq (n - i)c$. El tiempo requerido por la i-ésima pasada del bucle externo está acotado superiormente por $b + t(i)$ para una constante adecuada b, que tiene en cuenta las operaciones elementales efectuadas antes y después del bucle interno, y también el control del bucle para el bucle externo. Por tanto, el tiempo total invertido por el algoritmo está acotado superiormente por

$$\begin{aligned}\sum_{i=1}^{n-1} b + (n-i)c &= \sum_{i=1}^{n-1}(b+cn) - c\sum_{i=1}^{n-1} i \\ &= (n-1)(b+cn) - cn(n-1)/2 \\ &= \frac{1}{2}cn^2 + \left(b - \frac{1}{2}c\right)n - b,\end{aligned}$$

que está en $O(n^2)$. Un razonamiento similar muestra que este tiempo también está en $\Omega(n^2)$ en todos los casos, y por tanto la ordenación por selección requiere un tiempo que está en $\Theta(n^2)$ para ordenar n objetos.

El argumento anterior se puede simplificar, obviando la necesidad de introducir constantes explícitas tales como a, b y c, una vez que estemos habituados a la noción de instrucción barómetro. Aquí resulta natural tomar como barómetro la comprobación del bucle más interno, "**si** $T[j]<minx$" como medida del tiempo total de ejecución del algoritmo, porque ninguno de los bucles se puede ejecutar cero veces (en cuyo caso el control del bucle sería más oneroso en términos de tiempo que nuestro barómetro). Se verá fácilmente que el número de veces que se ejecuta la comprobación es:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1}(n-i) \\ &= \sum_{k=1}^{n-1} k = n(n-1)/2\end{aligned}$$

Por tanto el número de veces que se ejecuta la instrucción barómetro está en $\Theta(n^2)$, lo cual nos da automáticamente el tiempo de ejecución del algoritmo en sí.

Ordenación por inserción

También encontramos la ordenación por inserción en la Sección 2.4:

```
procedimiento inserción(T[1..n])
  para i ← 2 hasta n hacer
    x ← T[i]; j ← i - 1
    mientras j > 0 y x < T[j] hacer   T[j + 1] ← T[j]
                                      j ← j - 1
    T[j + 1] ← x
```

A diferencia de la ordenación por selección, se vio en la Sección 2.4 que el tiempo necesario para ordenar n objetos por inserción depende significativamente del orden original de los elementos. Aquí analizamos este algoritmo en el caso peor; el análisis del caso medio se da en la Sección 4.5. Para analizar el tiempo de ejecución de este algoritmo, seleccionamos como barómetro el número de veces que se comprueba la condición ($j > 0$ **y** $x < T[j]$) del bucle **mientras**.

Consideremos un valor fijo de i. Sea $x = T[i]$, como en el algoritmo. El caso peor surge cuando x es menor que $T[j]$ para todo j entre 1 e i–1, puesto que en este caso tenemos que comparar x con $T[i-1]$, $T[i-2]$,...,$T[1]$ antes de poder salir del bucle **mientras**, porque $j = 0$. Por tanto el bucle **mientras** se efectúa i veces en el caso peor. Este caso peor sucede para todos los valores de i entre 2 y n cuando el vector está ordenado inicialmente por orden decreciente. Entonces la comprobación del barómetro se efectúa $\Sigma_{i=2}^{n} i = n(n+1)/2-1$ veces en total, lo cual está en $\Theta(n^2)$. Esto muestra que la ordenación por inserción requiere también un tiempo que está en $\Theta(n^2)$ para ordenar n objetos en el caso peor.

Algoritmo de Euclides

Se recordará de la Sección 2.7.4 que el algoritmo de Euclides se utilizaba para calcular el máximo común divisor de dos enteros.

```
función Euclid(m, n)
   mientras m > 0 hacer
      t ← m
      m ← n mod m
      n ← t
   devolver n
```

El análisis de este bucle es ligeramente más sutil que los que hemos visto hasta el momento, porque está claro que no hay un progreso medible en cada pasada por el bucle, sino más bien en ocasiones alternativas. Para ver esto, mostraremos primero que para dos enteros cualesquiera m y n tales que $n \geq m$, siempre es cierto que $n \bmod m < n/2$:

- Si $m > n/2$, entonces $1 \leq n/m < 2$, y por tanto $n \div m = 1$, lo cual implica que $n \bmod m = n - m \times (n \div m) = n - m < n - n/2 = n/2$
- Si $m \leq n/2$, entonces $n \bmod m < m \leq n/2$

Suponemos sin pérdida de generalidad que $n \geq m$ puesto que en caso contrario la primera pasada por el bucle intercambia n y m (porque $n \bmod m = n$ cuando $n < m$). Esta condición se mantiene en todas las pasadas por el bucle, porque $n \bmod m$ nunca es mayor que m. Si suponemos que las operaciones aritméticas son elementales, lo cual es razonable en muchas situaciones, entonces el tiempo

total requerido por el algoritmo es del orden exacto del número de pasadas por el bucle. Determinemos una cota superior para este número como función de n. Considérese lo que sucede con m y n después de pasar *dos veces* por el bucle, suponiendo que el algoritmo no se detenga antes. Supongamos que m_0 y n_0 denotan el valor original de los parámetros. Después de la primera pasada por el bucle, m pasa a ser n_0 mod m_0. Después de la segunda pasada por el bucle, n toma ese valor. Por la observación anterior, n se ha vuelto más pequeño que $n_0/2$. En otras palabras, el valor de n se divide como mínimo por dos después de pasar dos veces por el bucle. En ese momento sigue siendo cierto que $n \geq m$ y por tanto vuelve a ser aplicable el mismo razonamiento: si el algoritmo no se ha detenido antes, otras dos pasadas por el bucle harán que el valor de n sea al menos dos veces más pequeño. Con cierta experiencia, la conclusión es ahora inmediata: en el bucle se va a entrar como máximo, aproximadamente, 2 lg n veces.

Formalmente, lo mejor es completar el análisis del algoritmo de Euclides tratando al bucle **mientras** como si fuese un algoritmo recursivo. Sea $t(\ell)$ el máximo número de veces que el algoritmo pasa por el bucle con unas entradas m y n cuando es $m \leq n \leq \ell$. Si $n \leq 2$, pasamos por el bucle o bien cero veces (si $m = 0$) o bien una vez. En caso contrario, o bien pasamos por el bucle menos de dos veces (si $m = 0$ ó m divide a n exactamente), o al menos dos veces. En este último caso, el valor de n se divide al menos por dos (y por tanto pasa a ser como mucho $\ell \div 2$) y el de m pasa a no ser mayor que el nuevo valor de n. Por tanto, se necesitan como máximo $t(\ell \div 2)$ pasadas adicionales por el bucle para completar el cálculo. Esto produce la recurrencia siguiente:

$$t(\ell) = \begin{cases} 1 & \text{si } \ell \leq 2 \\ 2 + t(\ell \div 2) & \text{en caso contrario} \end{cases}$$

Esto es un caso especial de la Ecuación 4.2 y las técnicas de la Sección 4.7 se aplican igualmente bien, para concluir que $t(\ell) \in O(\log \ell)$, lo cual demuestra que el algoritmo de Euclides se ejecuta en un tiempo lineal con respecto al *tamaño* de la entrada proporcionada, siempre y cuando sea razonable considerar como elementales todas las operaciones aritméticas. Es interesante saber que este algoritmo tiene su peor rendimiento con entradas de cualquier tamaño dado cuando m y n son dos números consecutivos de la sucesión de Fibonacci.

Quizá se pregunte el lector por qué hemos usado $t(\ell)$ para acotar el número de veces que pasa el algoritmo por el bucle con unas entradas m y n cuando $m \leq n \leq \ell$, en lugar de definir t directamente como una función de n, el mayor de los dos operandos. Parecería más natural definir $t(n)$ como el máximo número de veces que pasa el algoritmo por el bucle con unas entradas m y n cuando $m \leq n$. El problema es que esta definición no nos permitiría concluir que $t(n) \leq 2 + t(n \div 2)$ partiendo del hecho de que n se divide cuando menos por dos después de hacer dos pasadas por el bucle. Por ejemplo, $t(13) = 5$ con *Euclid*(8, 13) como caso peor, mientras que $t(13 \div 2) = t(6) = 2$. Esto sucede porque dos pasadas por el bucle después de una llamada a *Euclid*(8, 13)

no nos deja con $n = 6$ sino con $n = 5$ (y $m = 3$), y en efecto $t(13) \leq 2 + t(5)$ porque $t(5) = 3$ (precisamente con *Euclid*(3, 5) como caso peor). El origen del problema es que la definición "más natural" para t no proporciona una función no decreciente puesto que $t(5) > t(6)$, y por tanto la existencia de algún $n' \leq n \div 2$ tal que $t(n) \leq 2 + t(n')$ no implica que $t(n) \leq 2 + t(n \div 2)$. En lugar de ello, lo único que podemos decir es que $t(n) \leq 2 + \text{máx}\{t(n') \mid n' + n \div 2\}$, que es una recurrencia con la que resulta incómodo trabajar. Para resumir, hemos definido la función t precisamente en la forma en que lo hicimos para que sea evidentemente no decreciente, y sin embargo de tal modo que proporcione una cota superior del tiempo requerido para cualquier caso concreto.

Las Torres de Hanoi

Las Torres de Hanoi nos ofrecen otro ejemplo del análisis de algoritmos recursivos. Se dice que después de crear el mundo, Dios puso en la tierra tres barras hechas de diamantes y 64 anillos de oro. Estos anillos son todos ellos de distintos tamaños. En la creación, se dispusieron en la primera barra por orden de tamaños, con el mayor en la parte inferior y el menor en la parte superior. También creó Dios un monasterio junto a las barras. La tarea de los monjes durante su vida es trasladar todos los anillos a la segunda barra. La única operación permitida es trasladar un solo anillo de una barra a otra de tal manera que nunca se coloque un anillo encima de otro que sea menor. Cuando los monjes hayan finalizado su tarea, dice la leyenda, se acabará el mundo. Probablemente sea la profecía que ofrece más consuelo con respecto al fin del mundo, porque si los monjes se las arreglasen para mover un anillo por segundo, trabajando día y noche sin descansar ni cometer errores, ¡su trabajo no habría terminado 500.000 millones de años después de que empezaran! ¡Son más de 25 veces la edad estimada del Universo!

Es evidente que el problema se puede generalizar para un número arbitrario de anillos, n. Por ejemplo, con $n = 3$ obtenemos la solución dada en la figura 4.1. Para resolver el problema general, sólo tenemos que darnos cuenta de que para trasladar los m anillos más pequeños de la barra i a la barra j (en donde $1 \leq i \leq 3$, $1 \leq j \leq 3$, $i \neq j$ y $m \geq 1$), podemos trasladar primero los $m - 1$ anillos más pequeños de la barra i a la barra $k = 6 - i - j$, trasladamos después el m-ésimo anillo de la barra i a la barra j, y finalmente volvemos a trasladar los $m - 1$ anillos más pequeños de la barra k a la barra j. Lo que sigue es una descripción formal de este algoritmo; para resolver el caso original, lo único que hay que hacer (!) es invocarlo con los argumentos (64, 1,2):

```
procedimiento Hanoi(m, i, j)
    {Traslada los m anillos más pequeños de la barra i a la barra j}
    si m > 0 entonces Hanoi(m - 1, i, 6 - i - j)
                      escribir i " → "j
                      Hanoi(m - 1, 6 - i - j, j)
```

Para analizar el tiempo de ejecución de este algoritmo, utilizamos la instrucción **escribir** como barómetro. Supongamos que $t(m)$ denota el número de veces que se

ejecuta en una llamada a *Hanoi*(m, ·, ·). Por inspección del algoritmo, obtenemos la recurrencia siguiente:

$$t(m) = \begin{cases} 0 & \text{si } m = 0 \\ 2t(m-1)+1 & \text{en caso contrario} \end{cases} \tag{4.3}$$

a partir de la cual la técnica de la Sección 4.7 produce $t(m)=2^m - 1$; véase el Ejemplo 4.7.6. Dado que el número de ejecuciones de la instrucción **escribir**[4] es una buena medida del tiempo requerido por el algoritmo, concluimos que se necesita un tiempo que está en $\Theta(2^n)$ para resolver el problema con n anillos. De hecho, se puede demostrar que el problema con n anillos no se puede resolver en menos de 2^n–1 movimientos y por tanto el algoritmo es óptimo si uno insiste en imprimir toda la secuencia de movimientos.

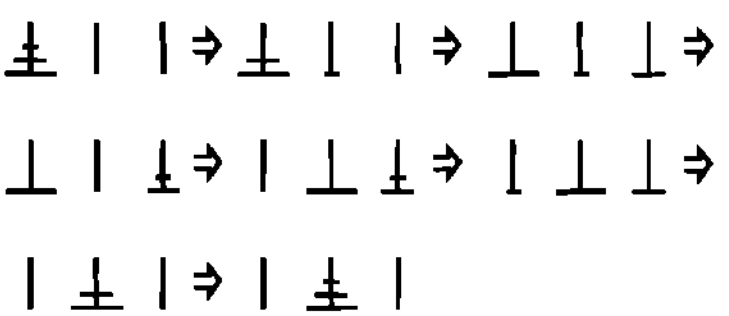


Figura 4.1. Las torres de Hanoi

Cálculo de determinantes

Otro ejemplo mas de análisis de la recursividad es el que concierne al algoritmo recursivo para calcular un determinante. Recuérdese que el determinante de una matriz $n \times n$ se puede calcular a partir de los determinantes de n matrices más pequeñas, $(n-1) \times (n-1)$ que se obtienen borrando la primera fila y alguna columna de la matriz original. Una vez que se han calculado los n subdeterminantes, el determinante de la matriz original se calcula con mucha rapidez. Además de las llamadas recursivas, la operación dominante que se necesita consiste en crear las n submatrices cuyos determinantes es preciso calcular. Esto requiere un tiempo que está en $\Theta(n^3)$ si se implementa directamente, pero basta un tiempo $\Theta(n)$ si se utilizan punteros en lugar de copiar elementos. Por tanto, el tiempo total $t(n)$ que se necesita para calcular el determinante de una matriz $n \times n$ mediante el algoritmo recursivo está dado por la recurrencia $t(n) = nt(n\text{-}1) + h(n)$ para $n \geq 2$, donde $h(n) \in \Theta(n)$. Esta recurrencia no se puede tratar mediante las técnicas de la Sección 4.7. Sin embargo, vimos en el problema 1.31 que la inducción constructiva es aplicable para concluir que $t(n) \in \Theta(n!)$, lo cual muestra que este algoritmo es muy ineficiente. La misma conclusión es válida si somos menos inteligentes y necesitamos $h(n) \in \Theta(n^3)$. Recuérdese de la Sección 2.7.1 que el determinante se puede calcular en un tiempo que está en $\Theta(n^3)$ por eliminación de Gauss-Jordan, e incluso más deprisa mediante *otro* algoritmo recursivo de la familia "divide y vencerás". Se precisa más esfuerzo para analizar estos algoritmos si se tiene en cuenta el tiempo requerido para las operaciones aritméticas.

4.5 ANÁLISIS DEL CASO MEDIO

Vimos que la ordenación por inserción requiere un tiempo cuadrático en el caso peor. Por otra parte, es fácil mostrar que concluye en un tiempo lineal en el caso *mejor*. Es natural preguntarse por su eficiencia *en el caso medio*. Para que esta pregunta tenga sentido, debemos ser precisos acerca del significado de "en el caso medio". Esto requiere suponer *a priori* una distribución de probabilidad para los casos en que se pedirá que resuelva nuestro algoritmo. La conclusión del análisis en el caso medio puede depender crucialmente de esta suposición, y este análisis puede inducir a error si de hecho nuestras suposiciones no se corresponden con la realidad de la aplicación que utiliza el algoritmo. Este tema tan importante se discutía con más extensión en la Sección 2.4, y volveremos a él en la Sección 10.7. En la mayoría de las ocasiones, los análisis en el caso medio se efectúan haciendo la suposición (más o menos realista) consistente en que todos los ejemplares de un tamaño dado son igualmente probables. Para los problemas de ordenación, resulta más sencillo suponer también que todos los elementos que hay que ordenar son distintos.

Supongamos que se tienen que ordenar n elementos distintos por inserción, y que las $n!$ permutaciones de estos elementos son igualmente probables. Para determinar el tiempo requerido en el caso medio por el algoritmo, podríamos sumar los tiempos requeridos para ordenar todas y cada una de las permutaciones posibles y dividir entonces por $n!$ la respuesta obtenida. Una aproximación alternativa, más sencilla en este caso, consiste en analizar directamente el tiempo requerido por el algoritmo, razonando probabilísticamente a medida que vayamos avanzando. Para todo i, con $2 \leq i \leq n$, considérese el subsector $T[1..i]$. El *rango parcial* de $T[i]$ se define como la posición que ocuparía si estuviera ordenado el subsector. Por ejemplo, el rango parcial de $T[4]$ en [3, 6, 2, 5, 1, 7, 4] es 3 porque $T[1..4]$, después de ordenarlo, es [2, 3, 5, 6]. Claramente, el rango parcial de $T[i]$ no depende del orden de los elementos del subsector $T[1..i-1]$. Es fácil demostrar que si todas las $n!$ permutaciones de $T[1..n]$ son igualmente probables, entonces el rango parcial de $T[i]$ tiene las mismas probabilidades de tomar cualquier valor entre 1 e i, independientemente de los valores de i.

Supongamos ahora que se fija un i, con $2 \leq i \leq n$, y que vamos a entrar en el bucle **mientras**. Una inspección del algoritmo muestra que el subsector $T[1..i-1]$ contiene los mismos elementos que antes de que se llamara al algoritmo, aunque ahora estén ordenados, y que $T[i]$ sigue teniendo su valor original, puesto que aún no ha sido desplazado. Por tanto el rango parcial de $T[i]$ tiene las mismas probabilidades de tomar cualquier valor entre 1 e i.

Sea k este rango parcial. Seleccionamos de nuevo como barómetro el número de veces que se comprueba la condición ($j > 0$ **y** $x < T[j]$) del bucle **mientras**. Por definición del rango parcial, y dado que $T[1..i-1]$ está ordenado, esta comprobación se efectúa exactamente $i - k + 1$ veces. Dado que todo valor de k entre 1 e i tiene una probabilidad de ocurrir igual a $1/i$, el número medio de veces que se efectúa la comprobación barómetro para cualquier valor dado de i es

$$c_i = \frac{1}{i}\sum_{k=1}^{i}(i-k+1) = \frac{i+1}{2}.$$

Estos sucesos son independientes para distintos valores de i. El número total medio de veces que se efectúa la comprobación barómetro cuando se ordenan n elementos es por tanto

$$\sum_{i=2}^{n} c_i = \sum_{i=2}^{n} \frac{i+1}{2} = \frac{(n-1)(n-4)}{4},$$

que está en $\Theta(n^2)$. Concluimos entonces que la ordenación por inserción hace en el caso medio aproximadamente la mitad de comparaciones que en el caso peor, pero este número sigue siendo cuadrático.

4.6 ANÁLISIS AMORTIZADO

En algunas ocasiones, el análisis del caso peor es excesivamente pesimista. Considérese por ejemplo un proceso P que tiene efectos secundarios, lo cual significa que P modifica el valor de variables globales. Como resultado de los efectos secundarios, dos llamadas sucesivas e idénticas a P podrían requerir cantidades de tiempo radicalmente diferentes. Cuando se esté analizando un algoritmo que tenga a P como subalgoritmo, lo fácil sería analizar P en el caso peor, y suponer que sucede lo peor cada vez que se llama a P. Este enfoque produce una respuesta correcta suponiendo que estemos satisfechos con un análisis en notación O, por oposición a la notación Θ, pero la respuesta sería pesimista. Considérese por ejemplo el bucle siguiente:

para $i \leftarrow 1$ **hasta** n **hacer** P

Si P requiere un tiempo en $\Theta(\log n)$ en el caso peor, es correcto concluir que el bucle requiere un tiempo en $O(n \log n)$, pero puede suceder que sea mucho más rápido *incluso en el caso peor*. Esto podría suceder si P no puede tardar mucho tiempo ($\Omega(\log n)$) a no ser que haya sido invocado muchas veces anteriormente, con un coste pequeño en todas las llamadas. Por ejemplo, podría suceder que P requiriera un tiempo constante en el caso medio, en cuyo caso el bucle completo se ejecutaría en un tiempo lineal.

En este caso, el significado de "en el caso medio" es completamente distinto del que se encontraba en la Sección 4.5. En lugar de tomar la media sobre todas las posibles entradas, lo cual requiere una suposición acerca de la distribución de probabilidad de los casos, tomamos la media sobre llamadas *sucesivas*. Aquí los tiempos requeridos por las diferentes llamadas tienen una fuerte relación entre sí, mientras que en la Sección 4.5 suponíamos implícitamente

que cada llamada era independiente de las demás. Para evitar confusiones, diremos en este contexto que toda llamada a P requiere un tiempo constante *amortizado* en lugar de decir que requiere un tiempo constante en el caso medio.

Decir que un proceso requiere un tiempo constante amortizado significa que existe una constante c (que sólo dependerá de la implementación) tal que para todo n positivo y para cualquier secuencia de n llamadas al proceso, el tiempo total para esas llamadas está acotado superiormente por cn. Por tanto, se permite un tiempo excesivo para una llamada sólo si se han registrado tiempos muy breves anteriormente, y *no* si tan sólo las llamadas posteriores fueran más rápidas. De hecho, si permitiéramos que una llamada fuera costosa basándonos en que nos prepara para llamadas posteriores mucho más rápidas, entonces el gasto se desperdiciaría si esa llamada fuera la última.

Considérese por ejemplo el tiempo necesario para conseguir una taza de café en la sala de profesores. La mayor parte de las veces basta entrar, coger la cafetera y poner el café en nuestra taza. De vez en cuando, sin embargo, tiene uno la mala suerte de encontrar vacía la cafetera: hay que hacer otra, lo cual requiere bastante más tiempo. Y ya de paso, puede uno hacer un poco de limpieza. Por tanto, el algoritmo para tomar café requiere una cantidad notable de tiempo en el caso peor, y sin embargo es rápido en el sentido amortizado, porque sólo se tarda mucho tiempo después de haber obtenido rápidamente bastantes tazas. (Para que esta analogía funcione correctamente, es preciso asumir con muy poco realismo que la cafetera está llena cuando entra la primera persona; en caso contrario precisamente la primera taza requeriría demasiado tiempo.)

Un ejemplo clásico de este comportamiento en programación concierne a la reserva de espacio con una necesidad ocasional de "recogida de basura". Un ejemplo más sencillo es el que concierne a la actualización de un contador binario. Supongamos que se desea manejar el contador como si fuera un vector de *bits* que representan el valor del contador en notación binaria: el vector $C[1..m]$ representa $\sum_{j=1}^{m} 2^{m-j}C[j]$. Por ejemplo, el vector [0, 1, 1, 0, 1, 1] representa el 27. Dado que uno de estos contadores sólo puede llegar hasta $2^m - 1$, supondremos que nos basta con contar el módulo 2^m (alternativamente, podríamos generar un mensaje de error en caso de desbordamiento). Lo que sigue es el algoritmo para añadir 1 al contador:

```
procedimiento contar (C[1..m])
    {Este procedimiento supone que m ≥ 1
    y que C[j] ∈ {0, 1} para todo 1 ≤ j ≤ m}
    j ← m + 1
    repetir
        j ← j – 1
        C[j] ← 1 – C[j]
    hasta_que C[j] = 1 o j = 1
```

Si se invoca con el ejemplo anterior [0, 1, 1, 0, 1, 1], el vector pasa a ser [0, 1, 1, 0, 1, 0] en la primera pasada por el bucle; [0, 1, 1, 0, 0, 0] en la segunda pasada, y [0, 1, 1, 1, 0, 0] en la tercera (lo cual representa ciertamente el valor 28 en binario); el bucle termina con $j = 4$, puesto que ahora $C[4]$ es igual a 1. Claramente, el caso peor del bucle se produce cuando $C[j] = 1$ para todo j, en cuyo caso pasa por el bucle m veces. Por tanto, n llamadas a *contar* partiendo de un vector que contenga sólo ceros va a requerir un tiempo total $O(nm)$. ¿Pero se necesita un tiempo en $\Theta(nm)$? La respuesta es negativa, por cuanto vamos a mostrar que *contar* requiere un tiempo amortizado constante. Esto implica que nuestras n llamadas a *contar* en conjunto requieren un tiempo en $\Theta(n)$, con una constante oculta que no depende de m. En particular, se puede conseguir contar desde 0 hasta $n = 2^m-1$ en un tiempo lineal en n, mientras que un análisis descuidado de *contar* en el caso peor produciría la conclusión correcta pero pesimista de que se necesita un tiempo que está en $O(n \log n)$.

Para alcanzar los resultados de un análisis amortizado, hay dos técnicas principales: el enfoque de la función potencial y el truco de contabilidad. La mejor aplicación de estas dos técnicas corresponde al análisis del número de veces que se ejecuta una instrucción barómetro.

Funciones potenciales

Supongamos que el proceso que hay que analizar modifica una base de datos, y que la eficiencia de tal proceso cada vez que se invoca depende del estado actual de la base de datos. Asociamos con la base de datos una noción de "organización", conocida con el nombre de *función potencial* de la base de datos. Se permite que las llamadas al proceso requieran más tiempo que el valor medio siempre y cuando mejoren la organización de la base de datos. A la inversa, se permite que las llamadas rápidas la desorganicen. ¡Esto es precisamente lo que ocurría en la sala de profesores! La analogía tiene todavía más validez: cuanto más rápidamente llenemos la taza, más probable es que se salga el café, lo cual a su vez significa que se necesitará más tiempo cuando llegue el momento de limpiar. De forma similar, cuanto más rápido va el proceso cuando va deprisa, más se desorganiza la base de datos, lo cual a su vez requiere más tiempo cuando se hace necesaria una reorganización.

Formalmente, introduciremos una función potencial entera Φ del estado de la base de datos. Los valores más elevados de Φ se corresponden con estados más desorganizados. Sea ϕ_0 el valor de Φ en el estado inicial; representa nuestro estándar de organización. Sea ϕ_i el valor de Φ para la base de datos después de la i-ésima llamada al proceso, y sea t_i el tiempo necesario para esa llamada (o el número de veces que se efectúa la instrucción barómetro). Definimos el *tiempo amortizado* requerido por esa llamada en la forma $\hat{t}_i = t_i + \phi_i - \phi_{i-1}$.

De esta manera, $\hat{t}_i$ es el tiempo realmente necesario para efectuar la i-ésima llamada al proceso, más el incremento de potencial causado por esa llamada. A veces es mejor pensar en esto como el tiempo real menos el *de*-cremento de potencial, por cuanto esta expresión muestra que se permitirá que las operaciones que

organizan la base de datos se ejecuten durante más tiempo sin incurrir en una penalización en términos de tiempo amortizado.

Supongamos que T_n denota el tiempo total requerido por las n primeras llamadas al proceso, y denotemos mediante $\hat{T}_n$ el tiempo total *amortizado*.

$$\begin{aligned}\hat{T}_n &= \sum_{i=1}^{n} \hat{t}_i = \sum_{i=1}^{n} (t_i + \phi_i - \phi_{i-1}) \\ &= \sum_{i=1}^{n} t_i + \sum_{i=1}^{n} \phi_i - \sum_{i=1}^{n} \phi_{i-1} \\ &= T_n + \phi_n + \phi_{n-1} + \ldots + \phi_1 \\ &\qquad - \phi_{n-1} - \ldots - \phi_1 - \phi_0 \\ &= T_n + \phi_n - \phi_0\end{aligned}$$

Por tanto,

$$T_n = \hat{T}_n - (\phi_n - \phi_0)$$

Lo significativo de esto es que $T_n \leq \hat{T}_n$ es válido para todo n siempre y cuando ϕ_n nunca llegue a ser menor que ϕ_0. En otras palabras, el tiempo total amortizado es siempre una cota superior del tiempo total real necesario para efectuar una secuencia de operaciones, con tal de que nunca se permita que la base de datos quede más "organizada" que en su estado inicial (¡esto demuestra que una excesiva organización puede ser perjudicial!). Este enfoque resulta interesante cuando los tiempos reales varían de forma significativa entre una llamada y la siguiente, mientras que el tiempo amortizado es casi invariante. Por ejemplo, una secuencia de operaciones requiere un tiempo lineal cuando el tiempo amortizado por operación es constante, independientemente del tiempo real requerido para cada operación.

El reto, cuando uno intenta aplicar esta técnica, consiste en determinar la función potencial adecuada. Ilustraremos esto con nuestro ejemplo del contador binario. Una llamada a *contar* resulta cada vez más costosa a medida que el cero del extremo derecho del contador se va encontrando más hacia la izquierda. Por tanto, la función potencial que se nos viene a la mente inmediatamente sería m menos el mayor j tal que $C[j] = 0$. Resulta, sin embargo, que esta selección función potencial no es adecuada, porque una única operación puede desordenar arbitrariamente el contador (al añadir 1 al contador que representa 2^k–2 se da lugar a que esta función potencial salte de 0 a k). Afortunadamente, hay una función potencial más sencilla que funciona bien: se define $\Phi(C)$ como el número de bits iguales a 1 que hay en C. Claramente, nuestra condición de que nunca se permita al potencial disminuir más allá del potencial inicial es válida, puesto que el potencial inicial es cero.

¿Cuál es el coste amortizado de añadir 1 al contador, en términos del número de veces que pasemos por el bucle? Hay que considerar tres casos:

- Si el contador representa un entero par, entonces se pasa una sola vez por el bucle, porque se hace cambiar de 0 a 1 el *bit* del extremo derecho. Como resultado, hay un bit puesto a 1 más que antes. Por tanto, el coste real es una pasada por el bucle, y el incremento de potencial también es 1. Por definición, el coste amortizado de la operación es $1 + 1 = 2$.
- Si todos los *bits* del contador son iguales a 1, pasamos por el bucle m veces, cambiando a 0 todos los bits. Como resultado, el potencial pasa de m a 0. Por tanto, el coste amortizado es $m - m = 0$.
- En los casos restantes, cada vez que pasemos por el bucle decrementaremos el potencial por valor de 1, puesto que se pasa un bit de 1 a 0, salvo en la última pasada por el bucle, en la cual se incrementa el potencial en 1 porque se pasa un *bit* de 0 a 1. De esta manera, si pasamos k veces por el bucle, se decrementa $k - 1$ veces el potencial, y se incrementa una vez, con un decremento neto de $k - 2$. Por tanto, el coste amortizado es $k - (k - 2) = 2$.

En conclusión, el coste amortizado de añadir 1 a un contador binario es precisamente equivalente a pasar dos veces por el bucle, salvo que no hay ningún coste cuando el contador vuelve a cero. Dado que el coste real de una secuencia de operaciones nunca es mayor que el coste amortizado, esto demuestra que el número total de veces que hay que pasar por el bucle cuando se incrementa n veces sucesivas un contador es como mucho $2n$ siempre que el contador estuviera puesto a cero inicialmente. (Este análisis sería incorrecto si se tuviera en cuenta el tiempo necesario para poner a cero el contador, como si hubiésemos tenido en cuenta el tiempo necesario para preparar la primera cafetera por la mañana.)

El truco de contabilidad

Esta técnica se puede considerar como una remodelación del enfoque de la función potencial, pero resulta más fácil de aplicar en algunos contextos. La aplicación clásica del truco de contabilidad sirve para analizar la eficiencia de la estructura de partición que se describe en la Sección 5.9. El análisis real de estas estructuras, sin embargo, se omitirá por cuanto va más allá del alcance de este libro.

Supongamos que se ha conjeturado una cota superior τ del tiempo invertido en el sentido amortizado siempre que se llama al proceso P, y que se desea demostrar que nuestra intuición es correcta (τ podría depender de varios parámetros, tales como el tamaño del caso). Para utilizar el truco de contabilidad, es preciso establecer una cuenta bancaria virtual, que contiene cero fichas inicialmente. Cada vez que se invoca a P, se deposita en la cuenta un crédito de τ fichas. Cada vez que se ejecuta la instrucción barómetro, es preciso pagarla, gastando una ficha de la cuenta. La regla es que la cuenta nunca esté en números rojos. Esto asegura que sólo se permitan operaciones lentas si se ha producido ya un número suficiente de operaciones rápidas. Por tanto, basta mostrar que se cumple la regla para con-

cluir que el tiempo real invertido por cualquier secuencia de operaciones nunca va a sobrepasar el tiempo amortizado, y en particular que cualquier secuencia de s operaciones requiere un tiempo que es como mucho τs.

Para analizar nuestro ejemplo de un contador binario, colocaremos dos fichas en la cuenta por cada llamada a *contar* (ésta es nuestra suposición inicial), y gastaremos una ficha cada vez que *contar* pase por el bucle. El punto clave, una vez más, concierne al número de *bits* que estén puestos a 1 en el contador. Dejaremos que el lector verifique que toda llamada a *contar* incrementa (o decrementa) la cantidad disponible en el banco por valor precisamente del incremento (o decremento) de *bits* puestos a 1 en el contador al que de lugar (a no ser que el contador se ponga a cero, en cuyo caso se gastan menos fichas). En otras palabras, si había i bits puestos a 1 en el contador antes de la llamada y $j > 0$ *bits* posteriormente, entonces el número de fichas disponibles en la cuenta después de la llamada se habrá incrementado en $j - i$ (contando los incrementos negativos como decrementos). Consiguientemente, el número de fichas que hay en la cuenta es siempre exactamente igual al número de *bits* puestos a 1 en el contador en ese momento (salvo que el contador se haya puesto a cero, en cuyo caso habrá más fichas en la cuenta). Esto demuestra que la cuenta nunca queda en descubierto, puesto que el número de bits puestos a 1 no puede ser negativo, y por tanto toda llamada a *contar* costará como máximo dos fichas en el sentido amortizado. Utilizaremos esta técnica en la Sección 5.8 para analizar la eficiencia amortizada de los montículos binomiales.

4.7 RESOLUCIÓN DE RECURRENCIAS

El último paso indispensable cuando se está analizando un algoritmo es frecuentemente la resolución de una ecuación de recurrencia. Con un poco de experiencia y de intuición, la mayoría de las recurrencias se puede resolver mediante suposiciones inteligentes. Sin embargo, existe una potente técnica que se puede utilizar para resolver de forma casi automática ciertas clases de recurrencias. Ése es el tema principal de esta sección: la técnica de la *ecuación característica*.

4.7.1 Suposiciones inteligentes

Este enfoque suele desarrollarse en cuatro etapas: calcular los primeros valores de la recurrencia, buscar una regularidad, inventar una forma general adecuada y demostrar finalmente por inducción matemática (quizá por inducción constructiva) que esta forma es correcta.

Considérese la recurrencia siguiente:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 3T(n \div 2) + n & \text{en caso contrario} \end{cases} \tag{4.4}$$

Una de las primeras lecciones que le enseñará la experiencia si intenta resolver recurrencias es que las funciones discontinuas tales como la función trunca-

ción (implícita en $n \div 2$) son difíciles de analizar. Nuestro primer paso es sustituir $n \div 2$ por "$n/2$", que tiene mejor comportamiento, aplicando una restricción adecuada al conjunto de valores de n que consideremos inicialmente. Resulta tentador restringir n a valores pares, porque en este caso $n \div 2 = n/2$, pero la división recursiva de un número par por 2 puede producir un número impar mayor que 1. Por tanto, es mejor idea restringir n a una potencia de 2. Una vez que hayamos tratado este caso especial, el caso general se sigue fácilmente en notación asintótica a partir de la regla de suavidad dada en la Sección 3.4.

En primer lugar, tabulamos la recurrencia de las primeras potencias de 2:

n	1	2	4	8	16	32
$T(n)$	1	5	19	65	211	665

Todos los términos de esta tabla menos el primero se calculan a partir del término anterior. Por ejemplo, $T(16) = 3 \times T(8) + 16 = 3 \times 65 + 16 = 211$. ¿Pero es útil esta tabla? ¡Decididamente, en la secuencia no sigue un patrón evidente! ¿Qué regularidad podemos encontrar?

La solución se vuelve evidente si guardamos más "historia" acerca del valor de $T(n)$. En lugar de escribir $T(2) = 5$, resulta más útil escribir $T(2) = 3 \times 1 + 2$. Entonces,

$$T(4) = 3 \times T(2) + 4 = 3 \times (3 \times 1 + 2) + 4 = 3^2 \times 1 + 3 \times 2 + 4.$$

Continuamos de esta manera, escribiendo n como potencia explícita de 2:

n	$T(n)$
1	1
2	$3 \times 1 + 2$
2^2	$3^2 \times 1 + 3 \times 2^2$
2^3	$3^3 \times 1 + 3^2 \times 2 + 3 \times 2^2 + 2^3$
2^4	$3^4 \times 1 + 3^3 \times 2 + 3^2 \times 2^2 + 3 \times 2^3 + 2^4$
2^5	$3^5 \times 1 + 3^4 \times 2 + 3^3 \times 2^2 + 3^2 \times 2^3 + 3 \times 2^4 + 2^5$

Ahora la regla resulta evidente:

$$\begin{aligned} T(2^k) &= 3^k 2^0 + 3^{k-1} 2^1 + 3^{k-2} 2^2 + \ldots + 3^1 2^{k-1} + 3^0 2^k \\ &= \sum_{i=0}^{k} 3^{k-i} 2^i = 3^k \sum_{i=0}^{k} (2/3)^i \\ &= 3^k \times (1 - (2/3)^{k+1}) / (1 - 2/3) \quad \text{(Proposición 1.7.10)} \\ &= 3^{k+1} - 2^{k+1} \end{aligned} \tag{4.5}$$

Es fácil cotejar esta fórmula con nuestra tabulación anterior. Por inducción (no por inducción *matemática*), ahora estamos convencidos de que la ecuación 4.5 es correcta.

Sin embargo, vimos en la Sección 1.6 que no siempre hay que confiar en la inducción. Por tanto, el análisis de nuestra recurrencia cuando n es una potencia de 2 estará incompleto mientras no demostremos la ecuación 4.5 por inducción matemática. Este ejercicio sencillo se deja para el lector.

Volviendo la vista atrás, se podría haber adivinado la Ecuación 4.5 con sólo un poquito más de intuición. Para esto habría bastado tabular el valor de $T(n) + in$ para pequeños valores de i, tales como $-2 \leq i \leq 2$.

n	1	2	4	8	16	32
$T(n)-2n$	–1	1	11	49	179	601
$T(n)-n$	0	3	15	57	195	633
$T(n)$	1	5	19	65	211	665
$T(n)+n$	2	7	23	73	227	697
$T(n)-2n$	3	9	27	81	243	729

Esta vez resulta evidente de forma inmediata que $T(n) + 2n$ es una potencia exacta de 3, de lo cual se deduce fácilmente la ecuación 4.5.

¿Qué sucede cuando n no es una potencia de 2? Es bastante difícil resolver la recurrencia 4.4 exactamente. Afortunadamente, esto no es necesario si nos basta obtener la respuesta en notación asintótica. Para esto, conviene reescribir la Ecuación 4.5 en términos de $T(n)$, en lugar de hacerlo en términos de $T(2^k)$. Puesto que $n = 2^k$, se sigue que $k = \lg n$. Por tanto,

$$T(n) = T(2^{\lg n}) = 3^{1+\lg n} - 2^{1+\lg n}.$$

Empleando la propiedad $3^{\lg n} = n^{\lg 3}$ (que se comprueba fácilmente tomando el logaritmo binario de ambos lados de la ecuación; véase el problema 1.15) se sigue que

$$T(n) = 3n^{\lg 3} - 2n \tag{4.6}$$

cuando n es una potencia de 2. Utilizando la notación asintótica condicional, concluiremos que $T(n) \in \Theta(n^{\lg 3} \mid n$ es una potencia de 2). Dado que $T(n)$ es una función no decreciente (hecho que se demuestra fácilmente por inducción matemática) y que $n^{\lg 3}$ es una función suave, se sigue de la Sección 3.4 que $T(n) \in \Theta(n^{\lg 3})$ incondicionalmente.

En la práctica, nunca se necesitarán estas suposiciones para resolver una recurrencia tal como la ecuación 4.4, porque las técnicas que vamos a estudiar pueden resolverlas de forma automática en unos pocos minutos. Sin embargo, existen recurrencias para las cuales estas técnicas no resultan aplicables, y las suposiciones inteligentes siempre pueden aplicarse como último recurso.

4.7.2 Recurrencias homogéneas

Comenzaremos nuestro estudio de la *técnica de la ecuación característica* mediante la resolución de recurrencias homogéneas lineales con coeficientes constantes, esto es, recurrencias de la forma

$$a_0 t_n + a_1 t_{n-1} + \ldots . a_k t_{n-k} = 0 \qquad (4.7)$$

en donde los t_i son los valores que estamos buscando. Además de la Ecuación 4.7, se necesitan los valores de t_i correspondientes a k valores de i (normalmente con $0 \le i \le k-1$ ó $1 \le i \le k$) para determinar la secuencia. Estas *condiciones iniciales* se considerarán más adelante. Hasta entonces, lo normal es que la Ecuación 4.7 tenga infinitas soluciones. Esta recurrencia es:

◊ *lineal* porque no contiene términos de la forma $t_{n-i} \times t_{n\ i}$, t^2_{n-i}, y similares;
◊ *homogénea* porque la combinación lineal de los t_{n-i} es igual a cero, y
◊ *con coeficientes constantes* porque las a_i son constantes.

Considérese por ejemplo la recurrencia, ya familiar, correspondiente a la sucesión de Fibonacci:

$$f_n = f_{n-1} + f_{n-2}$$

La recurrencia se adapta fácilmente al esquema de la Ecuación 4.7, tras reescribirla en la forma evidente:

$$f_n - f_{n-1} - f_{n-2} = 0$$

Por tanto, la sucesión de Fibonacci se corresponde con una recurrencia homogénea lineal de coeficientes constantes con $k = 2$, $a_0 = 1$ y $a_1 = a_2 = -1$.

Antes de empezar siquiera a buscar soluciones de la Ecuación 4.7, resulta interesante observar que toda combinación lineal de soluciones es, a su vez, una solución. En otras palabras, si f_n y g_n satisfacen la Ecuación 4.7, entonces $\Sigma_{i=0}^{k} a_i f_{n-i} = 0$ y lo mismo sucede para g_n, y si ponemos $t_n = cf_n + dg_n$ para unas constantes arbitrarias c y d, entonces t_n también es una solución de la Ecuación 4.7. Esto es cierto porque

$$\begin{aligned}
&a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} \\
&= a_0(cf_n + dg_n) + a_1(cf_{n-1} + dg_{n-1}) + \ldots + a_k(cf_{n-k} + dg_{n-k}) \\
&= c(a_0 f_n + a_1 f_{n-1} + \ldots + a_k f_{n-k}) + d(a_0 g_n + a_1 g_{n-1} + \ldots + a_k g_{n-k}) \\
&= c \times 0 + d \times 0 = 0.
\end{aligned}$$

Esta regla se generaliza para combinaciones lineales de cualquier número de soluciones.

Cuando se intenta resolver algunos ejemplos sencillos de recurrencias de la forma de la Ecuación 4.7 (*no* la sucesión de Fibonacci) mediante suposiciones inteligentes, surge de modo natural buscar soluciones de la forma $t_n = x^n$, en donde x es una constante desconocida por el momento. Si probamos en la Ecuación 4.7 esta posible solución, obtenemos:

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k} = 0$$

Esta ecuación se satisface si $x = 0$, una solución trivial sin interés. En caso contrario, la ecuación se satisface si y sólo si

$$a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0$$

Esta ecuación de grado k en x se denomina *ecuación característica* de la recurrencia 4.7 y

$$p(x) = a_0 x^k + a_1 x^{k-1} + \ldots + a_k$$

se denominá su *polinomio característico.*

Recuerde que el teorema fundamental del álgebra afirma que todo polinomio $p(x)$ de grado k posee exactamente k raíces (no necesariamente distintas), lo cual significa que se puede factorizar como un producto de k monomios:

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

donde los r_i pueden ser números complejos. Además, estos r_i son las únicas soluciones de la ecuación $p(x) = 0$.

Considérese cualquier raíz r_i del polinomio característico. Dado que $p(r_i) = 0$, se sigue que $x = r_i$ es una solución de la ecuación característica, y por tanto r_i^n es una solución de la recurrencia. Dado que toda combinación lineal de soluciones es también una solución, concluimos que

$$t_n = \sum_{i=1}^{k} c_i r_i^n \tag{4.8}$$

satisface la recurrencia para cualquier selección de constantes $c_1, c_2, \ldots, c_k$. El hecho notable, que no demostraremos aquí, consiste en que la Ecuación 4.7 *solamente* posee soluciones de esta forma, *siempre y cuando todos los* r_i *sean distintos.* En este caso, las k constantes se pueden determinar a partir de k condiciones iniciales resolviendo un sistema de k ecuaciones lineales con k incógnitas.

Ejemplo 4.7.1. (Fibonacci). Considérese la recurrencia

$$f_n = \begin{cases} n & \text{si } n = 0 \text{ o } n = 1 \\ f_{n-1} + f_{n-2} & \text{en caso contrario} \end{cases}$$

Primero reescribimos esta recurrencia para que se adapte a la forma de la ecuación 4.7:

$$f_n - f_{n-1} - f_{n-2} = 0$$

El polinomio característico es

$$x^2 - x - 1$$

cuyas raíces son

$$r_1 = \frac{1+\sqrt{5}}{2} \qquad \text{y} \qquad r_2 = \frac{1-\sqrt{5}}{2}.$$

La solución general, por tanto, es de la forma

$$f_n = c_1 r_1^n + c_2 r_2^n. \tag{4.9}$$

Todavía hay que utilizar las condiciones iniciales para determinar las constantes c_1 y c_2. Cuando $n = 0$, la ecuación 4.9 produce $f_0 = c_1 + c_2$. Pero sabemos que $f_0 = 0$. Por tanto, $c_1 + c_2 = 0$. De forma similar, cuando $n = 1$, la ecuación 4.9 junto con la segunda condición inicial nos dice que $f_1 = c_1 r_1 + c_2 r_2 = 1$. Recordando que los valores de r_1 y r_2 son conocidos, esto nos da dos ecuaciones lineales con las dos incógnitas c_1 y c_2.

$$\begin{aligned} c_1 \quad + \quad c_2 \quad &= \quad 0 \\ r_1 c_1 \quad + \quad r_2 c_2 \quad &= \quad 1 \end{aligned}$$

Resolviendo estas ecuaciones, obtenemos

$$c_1 = \frac{1}{\sqrt{5}} \qquad \text{y} \qquad c_2 = -\frac{1}{\sqrt{5}}$$

Por tanto,

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right],$$

que es la famosa fórmula de Moivre para la sucesión de Fibonacci. Obsérvese la sencillez mucho mayor de la técnica de la ecuación característica con respecto al enfoque de inducción constructiva que se veía en la Sección 1.6.4. Y además es más precisa, puesto que lo único que se podía descubrir con la inducción constructiva es que "f_n crece exponencialmente con un valor próximo a ϕ"; ahora disponemos de una fórmula exacta.

Si le sorprende que la solución de una recurrencia con coeficientes y condiciones iniciales enteras contenga números irracionales, intente el problema 4.31 y se llevará una sorpresa aún mayor.

Ejemplo 4.7.2. Considérese la recurrencia

$$t_n = \begin{cases} 0 & \text{si } n = 0 \\ 5 & \text{si } n = 1 \\ 3t_{n-1} + 4t_{n-2} & \text{en caso contrario} \end{cases}$$

En primer lugar reescribimos la recurrencia:

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

El polinomio característico es

$$x^2 - 3x - 4 = (x + 1)(x - 4)$$

cuyas raíces son $r_1 = -1$ y $r_2 = +4$. Por tanto, la solución general es de la forma

$$t_n = c_1(-1)^n + c_2 4^n$$

Las condiciones iniciales nos dan

$$\begin{array}{rcrcll} c_1 & + & c_2 & = & 0 & n = 0 \\ -c_1 & + & 4c_2 & = & 5 & n = 1 \end{array}$$

Al resolver estas ecuaciones, obtenemos $c_1 = -1$ y $c_2 = 1$. Por tanto,

$$t_n = 4^n - (-1)^n.$$ ❑

La situación se vuelve ligeramente más complicada cuando el polinomio característico tiene raíces múltiples, esto es, cuando las k raíces no son todas diferentes. Sigue siendo cierto que la Ecuación 4.8 satisface la recurrencia para valores cualesquiera de las constantes c_i, pero *ésta ya no es la solución más general*. Para hallar otras soluciones, sea $p(x) = a_0x^k + a_1x^{k-1} + \ldots + a_k$ el polinomio característico de nuestra recurrencia, y sea r una raíz múltiple. Por definición de las raíces múltiples, existe un polinomio $q(x)$, de grado k–2, tal que $p(x) = (x\text{-}r)^2q(x)$. Para todo $n \geq k$, considérense los polinomios de grado n

$$u_n(x) = a_0x^n + a_1x^{n-1} + \ldots + a_kx^{n-k} \text{ y}$$
$$v_n(x) = a_0nx^n + a_1(n-1)x^{n-1} + \ldots + a_k(n-k)x^{n-k}.$$

Observe que $v_n(x) = x \times u'_n(x)$, en donde $u'_n(x)$ denota la derivada de $u_n(x)$ con respecto a x. Ahora bien, $u_n(x)$ se puede reescribir en la forma

$$u_n(x) = x^{n-k}p(x) = x^{n-k}(x-r)^2 q(x) = (x-r)^2 \times \left[x^{n-k}q(x)\right].$$

Utilizando la regla para calcular la derivada de un producto de funciones, obtenemos que la derivada de $u_n(x)$ con respecto a x es

$$u_n'(x) = 2(x-r).x^{n-k}q(x) + (x-r)^2 \text{ x } \left[x^{n-k}q(x)\right]'.$$

Por tanto, $u_n'(r) = 0$, lo cual implica que $v_n(r) = r \times u_n'(r) = 0$ para todo $n \geq k$. En otras palabras, que

$$a_0 nr^n + a_1(n-1)r^{n-1} + \ldots + a_k(n-k)r^{n-k} = 0.$$

Concluimos que $t_n = nr^n$ es también una solución de la recurrencia. Se trata de una solución verdaderamente nueva, en el sentido de que no se puede obtener seleccionando unas constantes c_i adecuadas en la Ecuación 4.8.

Con más generalidad, si la raíz r tiene multiplicidad m, entonces $t_n = r^n$, $t_n = nr^n$, $t_n = n^2r^n$, ..., $t_n = n^{m-1}r^n$, son todas ellas soluciones diferentes de la recurrencia. La solución general es una combinación lineal de estos términos, y de los términos con que contribuyan las demás raíces del polinomio característico. Para resumir, si r_1, r_2, ..., r_l son las raíces distintas del polinomio característico, y sus multiplicidades son m_1, m_2, ..., m_l respectivamente, entonces

$$t_n = \sum_{i=1}^{l} \sum_{j=0}^{m_i - 1} c_{ij} n^j r_i^n$$

es la solución general de la Ecuación 4.7. Una vez más, las constantes c_{ij}, con $1 \leq i \leq l$ y $0 \leq j \leq m_i - 1$ tienen que determinarse a partir de las k condiciones iniciales. Hay k de estas constantes porque $\Sigma_{i=1}^{l} m_i = k$ (la suma de las multiplicidades de las diferentes raíces es igual al número total de raíces). Por sencillez, rotularemos normalmente las constantes en la forma c_1, c_2, ..., c_k, en lugar de utilizar dos índices.

Ejemplo 4.7.3. Considérese la recurrencia

$$t_n = \begin{cases} n & \text{si } n = 0,\ 1 \text{ ó } 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{en caso contrario} \end{cases}$$

Primero reescribimos la recurrencia:

$$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$$

El polinomio característico es

$$x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2$$

Las raíces, por tanto, son $r_1 = 1$ (de multiplicidad $m_1 = 1$) y $r_2 = 2$ (de multiplicidad $m_2 = 2$), y la solución general es

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n.$$

Las condiciones iniciales nos dan

$$\begin{array}{lllll} c_1 + c_2 & & = 0 & n = 0 \\ c_1 + 2c_2 + 2c_3 & & = 1 & n = 1 \\ c_1 + 4c_2 + 8c_3 & & = 2 & n = 2 \end{array}$$

Resolviendo estas ecuaciones, obtenemos $c_1 = -2$, $c_2 = 2$ y $c_3 = -\frac{1}{2}$. Por tanto:

$$t_n = 2^{n+1} - n2^{n-1} - 2$$

4.7.3 Recurrencias no homogéneas

La solución de una recurrencia lineal con coeficientes constantes se vuelve más difícil cuando la recurrencia no es homogénea, esto es, cuando la combinación lineal no es igual a cero. En particular, ya no es cierto que toda combinación lineal de las soluciones sea una solución. Comenzamos por un caso sencillo. Considérese la siguiente recurrencia:

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n) \tag{4.10}$$

El lado izquierdo es el mismo que antes, pero en el lado derecho tenemos $b^n p(n)$, en donde

◊ b es una constante, y

◊ $p(n)$ es un polinomio en n de grado d.

Ejemplo 4.7.4. Considérese la recurrencia

$$t_n - 2t_{n-1} = 3^n. \tag{4.11}$$

En este caso, $b = 3$ y $p(n) = 1$, un polinomio de grado 0. Un poco de astucia nos permitirá reducir este ejemplo al caso homogéneo con el que ya estamos familiarizados. Para ver esto, multiplicamos primero la recurrencia por 3, para obtener

$$3t_n - 6t_{n-1} = 3^{n+1}.$$

Ahora sustituimos n por n -1 en esta recurrencia, para obtener

$$3t_{n-1} - 6t_{n-2} = 3^n \tag{4.12}$$

Por último, se resta la ecuación 4.12 de la 4.11, para obtener

$$t_n - 5t_{n-1} + 6t_{n-2} = 0, \tag{4.13}$$

que se puede resolver mediante el método de la Sección 4.7.2. El polinomio característico es

$$x^2 - 5x + 6 = (x-2)(x-3)$$

y por tanto todas las soluciones son de la forma

$$t_n = c_1 2^n + c_2 3^n \tag{4.14}$$

Sin embargo, ya no es cierto que una elección arbitraria de las constantes c_1 y c_2 en la ecuación 4.14 produzca una solución de la recurrencia aun cuando no se tengan en cuenta las condiciones iniciales. Y hay algo peor: incluso las soluciones básicas $t_n = 2^n$ y $t_n = 3^n$, que desde luego son soluciones de la ecuación 4.13, no son soluciones de la recurrencia original dada por la ecuación 4.11. ¿Qué está pasando? La explicación es que las ecuaciones 4.11 y 4.13 *no* son equivalentes: la ecuación 4.13 se puede resolver dados valores arbitrarios para t_0 y t_1 (las condiciones iniciales), mientras que nuestra ecuación 4.11 original implica que $t_1 = 2t_0 + 3$. La solución general de la recurrencia original se puede determinar como una función de t_0, resolviendo las dos ecuaciones lineales en las incógnitas c_1 y c_2:

$$\begin{array}{rcrcll} c_1 & + & c_2 & = & t_0 & n = 0 \\ 2c_1 & + & 3c_2 & = & 2t_0 + 3 & n = 1 \end{array} \tag{4.15}$$

Resolviendo esto, se obtiene $c_1 = t_0 - 3$ y $c_2 = 3$. Por tanto, la solución general es

$$t_n = (t_0 - 3)2^n + 3^{n+1}$$

y de esta forma $t_n \in \Theta(3^n)$ independientemente de la condición inicial.

Siempre y cuando $t_0 \geq 0$, una sencilla demostración por inducción matemática basada en la ecuación 4.11 muestra que $t_n \geq 0$ para todo $n \geq 0$. Por tanto, resulta inmediato a partir de la ecuación 4.14 que $t_n \in O(3^n)$: no hay necesidad de resolver para las constantes c_1 y c_2 con objeto de alcanzar esta conclusión. Sin embargo, esta ecuación no basta por si misma para concluir que $t_n \in \Theta(3^n)$ porque *a priori* podría ser $c_2 = 0$. Sin embargo, resulta que el valor de c_2 se puede obtener directamente, sin necesidad de construir el sistema 4.15 de ecuaciones lineales. Esto basta para concluir que $t_n \in \Theta(3^n)$ independientemente del valor de t_0 (aun cuando sea negativo). Para ver esto, sustituimos la ecuación 4.14 en la recurrencia original:

$$\begin{aligned} 3^n &= t_n - 2t_{n-1} \\ &= (c_1 2^n + c_2 3^n) - 2(c_1 2^{n-1} + c_2 3^{n-1}) \\ &= c_2 3^{n-1} \end{aligned}$$

Independientemente de la condición inicial, concluimos que $c_2 = 3$ ❑

En los ejemplos siguientes, algunas veces preparamos un sistema de ecuaciones lineales para determinar todas las constantes que aparecen en la solución general, mientras que en otras ocasiones determinamos solamente las constantes necesarias para sustituir la solución general en la recurrencia original.

Ejemplo 4.7.5 Deseamos hallar la solución general de la siguiente recurrencia:

$$t_n - 2t_{n-1} = (n+5)3^n \qquad n \geq 1 \tag{4.16}$$

La manipulación necesaria para transformar esto en una recurrencia homogénea es algo más complicada que en el ejemplo 4.7.4. Es preciso:

a. escribir la recurrencia
b. sustituir en la recurrencia n por $n-1$ y multiplicar por –6, y
c. sustituir en la recurrencia n por $n-2$ y multiplicar por 9, obteniéndose sucesivamente:

$$\begin{array}{rcrcrcl} t_n & - & 2t_{n-1} & & & = & (n+5)3^n \\ & - & 6t_{n-1} & + & 12t_{n-2} & = & -6(n+4)3^{n-1} \\ & & & & 9t_{n-2} - 18t_{n-3} & = & 9(n+3)3^{n-2}. \end{array}$$

Sumando estas tres ecuaciones, obtenemos una recurrencia homogénea

$$t_n - 8t_{n-1} + 21t_{n-2} - 18t_{n-3} = 0$$

El polinomio característico es

$$x^3 - 8x^2 + 21x - 18 = (x-2)(x-3)^2$$

y por tanto todas las soluciones van a ser de la forma

$$t_n = c_1 2^n + c_2 3^n + c_3 n 3^n \tag{4.17}$$

Una vez más, toda selección de valores para las constantes c_1, c_2 y c_3 de la ecuación 4.17 proporciona una solución de la recurrencia homogénea, pero la recurrencia original impone restricciones sobre estas tres constantes porque requiere que $t_1 = 2t_0 + 18$ y $t_2 = 2t_1 + 63 = 4t_0 + 99$. Por tanto, la solución general se calcula resolviendo el siguiente sistema de ecuaciones:

$$\begin{array}{rcrcrcll} c_1 & + & c_2 & & & = & t_0 & n=0 \\ 2c_1 & + & 3c_2 & + & 3c_3 & = & 2t_0 + 18 & n=1 \\ 4c_1 & + & 9c_2 & + & 18c_3 & = & 4t_0 + 99 & n=2 \end{array}$$

Esto implica que $c_1 = t_0 - 9$, $c_2 = 9$ y $c_3 = 3$. Por tanto, la solución general de la ecuación 4.16 es

$$t_n = (t_0 - 9)2^n + (n + 3)3^{n+1}$$

y por tanto $t_n \in \Theta(n3^n)$ independientemente de la condición inicial.

Alternativamente, se podría sustituir la Ecuación 4.17 en la recurrencia original. Después de una manipulación sencilla, se obtiene:

$$(n+5)3^n = c_3 n3^{n-1} + (2c_3 + c_2)3^{n-1}$$

Igualando los coeficientes de $n3^n$ se tiene $1 = c_3/3$ y por tanto $c_3 = 3$. El hecho de que c_3 sea positivo basta para establecer el orden exacto de t_n, sin necesidad de resolver las demás constantes. Una vez que se conoce c_3, sin embargo, el valor de c_2 es igualmente sencillo de calcular por igualación de los coeficientes de $3n$: $c_2 = 15 - 2c_3 = 9$. ❑

Volviendo a los ejemplos 4.7.4 y 4.7.5, vemos que una parte del polinomio característico proviene del lado izquierdo de la ecuación 4.10, y el resto del lado derecho. La parte que proviene del lado izquierdo es exactamente igual que si la ecuación hubiera sido homogénea: $(x - 2)$ para ambos ejemplos. La parte que viene del lado derecho es un resultado de nuestra manipulación.

Generalizando, podemos mostrar que para resolver la ecuación 4.10 basta utilizar el siguiente polinomio característico.

$$(a_0x^k + a_1x^{k-1} + \ldots + \mathrm{a})(x - b)^{d+1}$$

Recuerde que d es el grado del polinomio $p(n)$. Una vez que se ha obtenido este polinomio, procedemos igual que en el caso homogéneo, salvo que algunas de las ecuaciones necesarias para determinar las constantes no se obtienen a partir de las condiciones iniciales sino a partir de la propia recurrencia en sí.

Ejemplo 4.7.6 El número de movimientos de un anillo que se necesita en el problema de las Torres de Hanoi (véase la Sección 4.4) está dado por la ecuación 4.3:

$$t(m) = \begin{cases} 0 & \text{si } m = 0 \\ 2t(m-1) + 1 & \text{en caso contrario} \end{cases}$$

La recurrencia se puede escribir en la forma

$$t(m) - 2t\,(m-1) = 1, \qquad (4.18)$$

que es la forma de la ecuación 4.10 con $b = 1$ y $p(n) = 1$, un polinomio de grado 0. El polinomio característico es, por tanto

$$(x - 2)(x - 1)$$

donde el factor $(x - 2)$ viene del lado izquierdo de la ecuación 4.18 y el factor $(x - 1)$ viene de su lado derecho. Las raíces del polinomio son 1 y 2, ambas de multiplicidad 1, así que todas las soluciones de esta recurrencia son de la forma

$$t(m) = c_1 1^m + c_2 2^m. \tag{4.19}$$

Necesitamos dos condiciones iniciales para determinar las constantes c_1 y c_2. Sabemos que $t_0 = 0$; para hallar la segunda condición inicial utilizamos la propia recurrencia para calcular

$$t(1) = 2t(0) + 1 = 1.$$

Esto nos da dos ecuaciones lineales en las constantes desconocidas:

$$\begin{aligned} c_1 + c_2 &= 0 \qquad m = 0 \\ c_1 + 2c_2 &= 1 \qquad m = 1 \end{aligned}$$

A partir de este sistema obtenemos la solución $c_1 = -1$ y $c_2 = 1$, y por tanto

$$t(m) = 2^m - 1.$$

Si lo único que deseamos determinar es el orden exacto de $t(m)$, no es necesario calcular las constantes de la ecuación 4.19. Esta vez no necesitamos ni siquiera sustituir la ecuación 4.19 en la recurrencia original. Basta saber que $t(m) = c_1 + c_2 2^m$ para concluir que $c_2 > 0$ y por tanto que $t(m) \in \Theta(2^m)$. Para esto, obsérvese que $t(m)$, el número de movimientos de un anillo que se necesitan, no puede ciertamente ser negativo ni una constante, puesto que claramente $t(m) \geq m$. ❑

Ejemplo 4.7.7. Considérese la recurrencia

$$t_n = 2t_{n-1} + n$$

Esto se puede reescribir en la forma

$$t_n - 2t_{n-1} = n$$

que es de la forma de la Ecuación 4.10 con $b = 1$ y $p(n) = n$, un polinomio de grado 1. El polinomio característico es entonces

$$(x-2)(x-1)^2$$

con raíces 2 (de multiplicidad 1) y 1 (de multiplicidad 2). Todas las soluciones son entonces de la forma

$$t_n = c_1 2^n + c_2 1^n + c_3 n 1^n \tag{4.20}$$

Siempre y cuando $t_0 \geq 0$, y por tanto $t_n \geq 0$ para todo n, concluimos inmediatamente que $t_n \in O(2^n)$. Se necesita un análisis posterior para afirmar que $t_n \in \Theta(2^n)$.

Si sustituimos la ecuación 4.20 en la recurrencia original, obtenemos

$$\begin{aligned} n &= t_n - 2t_{n-1} \\ &= (c_1 2^n + c_2 + c_3 n) - 2(c_1 2^{n-1} + c_2 + c_3(n-1)) \\ &= (2c_3 - c_2) - c_3 n \end{aligned}$$

de lo cual se obtiene directamente que $2c_3 - c_2 = 0$ y $-c_3 = 1$, independientemente de la condición inicial. Esto implica que $c_3 = -1$ y $c_2 = -2$. A primera vista, nos sentimos decepcionados porque es c_1 quien resulta ser relevante para determinar el orden exacto de t_n, según se ve en la ecuación 4.20, y lo que hemos obtenido son precisamente las otras dos constantes. Sin embargo, estas constantes transforman la ecuación 4.20 en

$$t_n = c_1 2^n - n - 2. \tag{4.21}$$

Siempre y cuando $t_0 \geq 0$ y por tanto $t_n \geq 0$ para todo n, la ecuación 4.21 implica que c_1 debe ser estrictamente positivo. Por tanto, tenemos derecho a concluir que $t_n \in \Theta(2^n)$ sin necesidad de resolver explícitamente para c_1. Por supuesto, c_1 se puede obtener ahora con facilidad a partir de la condición inicial, si se desea.

Alternativamente, se pueden determinar las tres constantes como funciones de t_n, preparando y resolviendo el oportuno sistema de ecuaciones lineales que se obtiene de la ecuación 4.20 y del valor de t_1 y t_2 calculados a partir de la recurrencia original. ❑

Llegados aquí, quizá piense que a efectos prácticos no tiene sentido preocuparse por las constantes: el orden exacto de t_n siempre se puede leer directamente a partir de la solución general. ¡No es así! O quizá piense que las constantes obtenidas por la técnica más sencilla de sustituir la solución general en la recurrencia original van a ser siempre suficientes para determinar su orden exacto. ¡Tampoco es así! Considere el ejemplo siguiente.

Ejemplo 4.7.8. Considérese la recurrencia

$$t_n = \begin{cases} 1 & \text{si } n = 0 \\ 4t_{n-1} - 2^n & \text{en caso contrario} \end{cases}$$

En primer lugar, se reescribe la recurrencia en la forma

$$t_n - 4t_{n-1} = -2^n,$$

que es la forma de la ecuación 4.10 con $b = 2$ y $p(n) = -1$, un polinomio de grado 0. Por tanto el polinomio característico es

$$(x - 4)(x - 2)$$

con raíces 4 y 2, ambas de multiplicidad 1. Todas las soluciones, por tanto, son de la forma

$$t_n = c_1 4^n + c_2 2^n \tag{4.22}$$

Quizá se sienta tentado de afirmar sin más que $t_n \in \Theta(4^n)$, puesto que éste es claramente el término dominante de la ecuación 4.22.

Si tiene menos prisa, quizá desee sustituir la ecuación 4.22 en la recurrencia original, para ver lo que sale:

$$\begin{aligned} -2^n &= t_n - 4t_{n-1} \\ &= c_1 4^n + c_2 2^n - 4(c_1 4^{n-1} + c_2 2^{n-1}) \\ &= -c_2 2^n \end{aligned}$$

Por tanto, $c_2 = 1$ independientemente de la condición inicial. El conocimiento de c_2 no es relevante directamente para determinar el orden exacto de t_n dado por la ecuación 4.22. A diferencia del ejemplo anterior, sin embargo, no se puede afirmar nada concluyente a partir del simple hecho de que c_2 sea positiva. Aun cuando hubiéramos averiguado que c_2 era negativa, seguiríamos sin poder concluir nada acerca de c_1 de forma inmediata, porque esta vez no hay razón evidente que nos haga suponer que t_n debe ser positivo.

Dado que ha fallado todo lo demás, nos vemos obligados a determinar las constantes. Podríamos preparar el sistema habitual de dos ecuaciones lineales con dos incógnitas que se obtiene a partir de la ecuación 4.22 y de los valores de t_0 y t_1, pero ¿por qué vamos a despreciar el conocimiento que ya tenemos acerca de c_2? Sabemos que $t_n = c_1 4^n + 2^n$. Sustituyendo la condición inicial $t_0 = 1$ se tiene $1 = c_1 + 1$ y por tanto $c_1 = 0$. La conclusión es que la solución exacta para esta recurrencia es simplemente $t_n = 2^n$, y que ¡la afirmación anterior, ($t_n \in \Theta(n^4)$, es incorrecta! Obsérvese sin embargo que t_n estaría en $\Theta(4^n)$ *s* se especificará como condición inicial al-

gún valor más grande de t_0, puesto que en general $c_1 = t_0 - 1$. Por otra parte, con una condición inicial $t_0 < 1$, t_n toma valores negativos que crecen con rapidez exponencial. Este ejemplo ilustra la importancia de la condición inicial para algunas recurrencias, mientras que los ejemplos anteriores habían mostrado que el comportamiento asintótico de muchas recurrencias no se ve afectado por la condición inicial, al menos cuando $t_0 \geq 0$. ❑

Una generalización adicional del mismo tipo de argumento nos permite resolver finalmente las recurrencias de la forma

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \ldots \tag{4.23}$$

donde los b_i son constantes distintas y los $p_i(n)$ son polinomios en n de grado d_i. Estas recurrencias se resuelven empleando el siguiente polinomio característico:

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1}\ldots,$$

que contiene un factor correspondiente al lado izquierdo y un factor correspondiente a cada término del lado derecho. Una vez que se ha obtenido el polinomio característico, la recurrencia se resuelve igual que antes.

Ejemplo 4.7.9. Considérese la recurrencia

$$t_n = \begin{cases} 0 & \text{si } n = 0 \\ 2t_{n-1} + n + 2^n & \text{en caso contrario} \end{cases}$$

En primer lugar, se reescribe la recurrencia en la forma

$$t_n - 2t_{n-1} = n + 2^n,$$

que es la forma de la ecuación 4.23 con $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$ y $p_2(n) = 1$. El grado de $p_1(n)$ es $d_1 = 1$ y el grado de $p_2(n)$ es $d_2 = 0$. El polinomio característico es

$$(x-2)(x-1)^2(x-2),$$

que tiene las raíces 1 y 2, ambas con multiplicidad 2. Todas las soluciones de la recurrencia, por tanto, tienen la forma

$$t_n = c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n. \tag{4.24}$$

A partir de esta ecuación, concluimos que $t_n \in O(n2^n)$ sin calcular las constantes, pero necesitamos saber si $c_4 > 0$ o no para determinar el orden exacto de

t_n. Para hacer esto, se sustituye la ecuación 4.24 en la recurrencia original, lo cual nos da

$$n + 2^n = (2c_2 - c_1) - c_2 n + c_4 2^n$$

Igualando los coeficientes de 2^n, se obtiene inmediatamente que $c_4 = 1$ y por tanto $t_n \in \Theta(n2^n)$. Las constantes c_1 y c_2 se pueden leer con igual facilidad si se desea. La constante c_3 se puede obtener a partir de la Ecuación 4.24, del valor de las demás constantes y de la condición inicial $t_0 = 0$.

Alternativamente, se pueden determinar las cuatro constantes resolviendo cuatro ecuaciones lineales en cuatro incógnitas. Dado que se necesitan cuatro ecuaciones y que sólo se tiene una ecuación inicial, utilizamos la recurrencia para calcular los valores de los otros tres puntos: $t_1 = 3$, $t_2 = 12$ y $t_3 = 35$. Esto da lugar al siguiente sistema:

$$\begin{array}{rcrcrcrcll}
c_1 & & & + & c_3 & & & = & 0 & n = 0 \\
c_1 & + & c_2 & + & 2c_3 & + & 2c_4 & = & 3 & n = 1 \\
c_1 & + & 2c_2 & + & 4c_3 & + & 8c_4 & = & 12 & n = 2 \\
c_1 & + & 3c_2 & + & 8c_3 & + & 24c_4 & = & 35 & n = 3
\end{array}$$

Al resolver este sistema se obtiene $c_1 = -2$, $c_2 = -1$, $c_3 = 2$ y $c_4 = 1$. Por tanto, obtenemos finalmente

$$t_n = n2^n + 2^{n+1} - n - 2$$

4.7.4 Cambios de variable

A veces es posible resolver recurrencias más complicadas efectuando un cambio de variable. En los ejemplos siguientes, escribiremos $T(n)$ para el término de una recurrencia general, y t_i para el término de una nueva recurrencia obtenida a partir de la primera mediante un cambio de variable. Asegúrese de estudiar el Ejemplo 4.7.13, que se encuentra entre las recurrencias más importantes a efectos de la Algorítmica.

Ejemplo 4.7.10 Reconsidere la recurrencia que se resolvió mediante suposiciones inteligentes en la Sección 4.7.1, pero sólo para el caso en el cual n es una potencia de 2.

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n/2) + n & \text{si } n \text{ es una potencia de 2, } n > 1 \end{cases}$$

Para traducila a una forma que sepamos resolver, sustituimos n por 2^i. Esto se consigue introduciendo una nueva recurrencia t_i, definida como $t_i = T(2^i)$. Esta

transformación resulta útil porque $n/2$ pasa a ser $(2^i)/2 = 2^{i-1}$. En otras palabras, nuestra recurrencia original en la que $T(n)$ se define como una función de $T(n/2)$ da lugar a una en la cual t_i está definido como una función de t_{i-1}, que es precisamente el tipo de recurrencia que hemos aprendido a resolver:

$$\begin{aligned} t_i &= T(2^i) = 3T(2^{i-1}) + 2^i \\ &= 3t_{i-1} + 2^i \end{aligned}$$

Una vez reescrita en la forma

$$t_i - 3t_{i-1} = 2^i$$

esta recurrencia tiene la forma de la Ecuación 4.10. El polinomio característico es

$$(x-3)(x-2)$$

y por tanto todas las soluciones para ti son de la forma

$$t_i = c_1 3^i + c_2 2^i.$$

Utilizamos ahora el hecho de que $T(2^i) = t_i$ y por tanto $T(n) = t_{\lg n}$ cuando $n = 2^i$ para obtener

$$\begin{aligned} T(n) &= c_1 3^{\lg n} + c_2 2^{\lg n} \\ &= c_1 n^{\lg 3} + c_2 n \end{aligned}$$

cuando n es una potencia de 2, lo cual basta para concluir que

$$T(n) \in O(n^{\lg 3} | n \text{ es una potencia de } 2)$$

Sin embargo, necesitamos mostrar que c_1 es estrictamente positivo antes de que podamos afirmar algo acerca del orden *exacto* de $T(n)$.

Ahora estamos familiarizados con dos técnicas para determinar las constantes. Para ejercitarlas, vamos a aplicar las dos a esta situación. La aproximación más directa, que no siempre proporciona la información deseada, consiste en sustituir la solución proporcionada por la ecuación 4.25 en la recurrencia original. Si tenemos en cuenta que $(1/2)^{\lg 3} = 1/3$, esto da lugar a

$$\begin{aligned} n &= T(n) - 3T(n/2) \\ &= (c_1 n^{\lg 3} + c_2 n) - 3(c_1 (n/2)^{\lg 3} + c_2 (n/2)) \\ &= -c_2 n/2 \end{aligned}$$

y por tanto $c_2 = -2$. Aun cuando no hemos obtenido el valor de c_1, que es la constante más relevante, estamos sin embargo en condiciones de afirmar que debe de ser estrictamente positiva, porque en caso contrario la ecuación 4.25 implicaría erróneamente que $T(n)$ es negativo. El hecho consistente en que

$$T(n) \in \Theta(n^{\lg 3} | n \text{ es una potencia de } 2) \tag{4.26}$$

queda por tanto demostrado. Por supuesto, ahora sería fácil obtener el valor de c_1 a partir de la Ecuación 4.25, del hecho consistente en que $c_2 = -2$, y de la condición inicial $T(1) = 1$, pero esto no es necesario si nos conformamos con resolver la recurrencia en notación asintótica. Además, hemos aprendido que la ecuación 4.26 es válida independientemente de la condición inicial, siempre y cuando $T(n)$ sea positivo.

El enfoque alternativo consiste en preparar dos ecuaciones lineales en las dos incógnitas c_1 y c_2. Se garantiza que producirá el valor de ambas constantes. Para esto necesitamos el valor de $T(n)$ en dos puntos. Ya sabemos que $T(1) = 1$. Para obtener otro punto, utilizamos la propia recurrencia: $T(2) = 3T(1) + 2 = 5$. Sustituyendo $n = 1$ y $n = 2$ en la ecuación 4.25, se tiene el sistema siguiente

$$\begin{array}{lcll} c_1 & + & c_2 = 1 & n = 1 \\ 3c_1 & + & 2c_2 = 5 & n = 2 \end{array}$$

Resolviendo estas ecuaciones, obtenemos $c_1 = 3$ y $c_2 = -2$. Por tanto:

$$T(n) = 3n^{\lg 3} - 2n$$

cuando n es una potencia de 2, lo cual desde luego es exactamente lo que se obtenía en la Sección 4.7.1 mediante suposiciones inteligentes. ❑

Ejemplo 4.7.11. Considérese la recurrencia

$$T(n) = 4T(n/2) + n^2$$

en donde n es una potencia de dos, y $n \geq 2$. Procederemos igual que en el ejemplo anterior:

$$\begin{aligned} t_i &= T(2^i) = 4T(2^{i-1}) + (2^i)^2 \\ &= 4t_{i-1} + 4^i \end{aligned}$$

Reescribimos esto en la forma de la ecuación 4.10:

$$t_i - 4t_{i-1} = 4^i$$

El polinomio característico es $(x–4)^2$ y por tanto todas las soluciones son de la forma

$$t_i = c_1 4^i + c_2 i 4^i$$

En términos de $T(n)$, esta ecuación es equivalente

$$T(n) = c_1 n^2 + c_2 n^2 \lg n. \tag{4.27}$$

Al sustituir la Ecuación 4.27 en la recurrencia original, tenemos

$$n^2 = T(n) - 4T(n/2) = c_2 n^2$$

y por tanto $c_2 = 1$. Es decir,

$$T(n) \in \Theta(n^2 \log n | n \text{ es una potencia de } 2),$$

independientemente de las condiciones iniciales (incluso si $T(1)$ es negativo). ❑

Ejemplo 4.7.12. Considérese la recurrencia

$$T(n) = 2T(n/2) + n \lg n$$

donde n es una potencia de 2, $n \geq 2$. Tal como antes, obtenemos

$$\begin{aligned} t_i &= T(2^i) = 2T(2^{i-1}) + i2^i \\ &= 2t_{i-1} + i2^i \end{aligned}$$

Reescribimos esto en la forma de la ecuación 4.10:

$$t_i - 2t_{i-1} = i2^i$$

El polinomio característico es $(x–2)(x–2)^2 = (x–2)^3$ y por tanto todas las soluciones son de la forma

$$t_i = c_1 2^i + c_2 i 2^i + c_3 i^2 2^i$$

En términos de $T(n)$, esta ecuación equivale

$$T(n) = c_1 n + c_2 n \lg n + c_3 n \lg^2 n. \tag{4.28}$$

Al sustituir la ecuación 4.28 en la recurrencia original, se obtiene

$$n \lg n = T(n) - 2T(n/2) = (c_2 - c_3)n + 2c_3 n \lg n,$$

lo cual implica que $c_2 = c_3$ y que $2c_3 = 1$, luego $c_2 = c_3 = \frac{1}{2}$. Por tanto:

$$T(n) \in \Theta(n \log^2 n | n \text{ es una potencia de } 2),$$

independientemente de las condiciones iniciales. ❑

Comentario: En los capítulos anteriores, la recurrencia dada para $T(n)$ sólo es aplicable cuando n es una potencia de 2. Por tanto, es inevitable que la solución obtenida deba estar en notación asintótica condicional. En todos los casos, sin embargo, es suficiente añadir la condición de que $T(n)$ sea asintéticamente no decreciente para poder concluir que los resultados asintóticos obtenidos son aplicables incondicionalmente a todos los valores de n. Esta condición se sigue de la regla de uniformidad (Sección 3.4) puesto que las funciones $n^{\lg 3}$, $n^2 \log n$ y $n \log^2 n$ son suaves.

Ejemplo 4.7.13. Ahora estamos preparados para resolver una de las recurrencias más importantes a efectos algorítmicos. Esta recurrencia es especialmente útil para el análisis de algoritmos de divide y vencerás, como se verá en el Capítulo 7. Las constantes $n_0 \geq 1$, $\ell \geq 1$, $b \geq 2$ y $k \geq 0$ son enteros, mientras que c es un número real estrictamente positivo. Sea $T: \mathbb{N} \to \mathbb{R}^+$ una función asintéticamente no decreciente tal que

$$T(n) = \ell T(n/b) + cn^k \qquad n > n_0 \tag{4.29}$$

cuando n/n_0 es una potencia exacta de b, esto es, cuando $n \in \{bn_0, b^2 n_0, b^3 n_0, \ldots\}$

Esta vez, el cambio de variable adecuado es $n = b^i n_0$.

$$\begin{aligned} t_i &= T(b^i n_0) = \ell T(b^{i-1} n_0) + c(b^i n_0)^k \\ &= \ell t_{i-1} + cn_0^k b^{ik} \end{aligned}$$

Reescribimos esto en la forma de la Ecuación 4.10.

$$t_i - \ell t_{i-1} = (cn_0^k)(b^k)^i$$

El lado derecho es de la forma deseada, $a^i p(i)$ en donde $p(i) = cn_0^k$ es un polinomio constante (de grado 0) y $a_0 = b^k$. Por tanto, el polinomio característico es $(x - \ell)(x - b^k)$, cuyas raíces son ℓ y b^k. A partir de esto, resulta tentador (¡pero es falso en general!) concluir que todas las soluciones son de la forma

$$t_i = c_1 \ell^i + c_2 (b^k)^i \tag{4.30}$$

Para escribir esto en términos de $T(n)$, obsérvese que $i = \log_b(n/n_0)$ cuando n es de la forma adecuada, y por tanto $d^i = (n/n_0)^{\log_b d}$ para valores positivos arbitrarios de d. Por tanto:

$$\begin{aligned} T(n) &= (c_1 / n_0^{\log_b \ell}) n^{\log_b \ell} + (c_2 / n_0^k) n^k \\ &= c_3 n^{\log_b \ell} + c_4 n^k \end{aligned} \tag{4.31}$$

para unas nuevas constantes adecuadas c_3 y c_4. Para conocer estas constantes, sustituimos la ecuación 4.31 en la recurrencia original:

$$\begin{aligned} cn^k &= T(n) - \ell T(n/b) \\ &= c_3 n^{\log_b \ell} + c_4 n^k - \ell (c_3 (n/b)^{\log_b \ell} + c_4 (n/b)^k) \\ &= \left(1 - \frac{\ell}{b^k}\right) c_4 n^k \end{aligned}$$

Por tanto, $c_4 = c / (1 - \ell / b^k)$. Para expresar $T(n)$ en notación asintótica, necesitamos mantener solamente el término dominante de la ecuación 4.31. Hay que considerar tres casos, dependiendo de si ℓ es menor, igual o mayor que b^k.

- ◊ Si $\ell < b_k$, entonces $c_4 > 0$ y $k > \log_b \ell$. Por tanto el término $c_4 n^k$ domina la ecuación 4.31. Concluimos que $T(n) \in \Theta(n^k \mid (n/n_0)$ es una potencia de $b)$. Pero n^k es una función suave, y $T(n)$ es asintéticamente no decreciente por hipótesis. Por tanto, $T(n) \in \Theta(n^k)$.
- ◊ Si $\ell > b^k$ entonces $c_4 < 0$ y $\log_b \ell > k$. El hecho de que c_4 sea negativo implica que c_3 es positivo, porque en caso contrario la ecuación 4.31 implicaría que $T(n)$ es negativo, lo cual va a en contra de la especificación $T: \mathbb{N} \rightarrow \mathbb{R}^+$. Por tanto el término $c_3 n^{\log_b \ell}$ domina la ecuación 4.31. Más aún $n^{\log_b \ell}$ es una función suave y $T(n)$ es eventualmente no decreciente. Por tanto $T(n) \in \Theta(n^{\log_b \ell})$.
- ◊ Si $\ell = b^k$, sin embargo, tenemos problemas, porque ¡la fórmula de c_4 implica una división por cero! Lo que ha ido mal en este caso es que el polinomio característico tiene una sola raíz de multiplicidad 2, en lugar de tener dos raíces distintas. Por tanto, la ecuación 4.30 *no* proporciona la solución general de la recurrencia. En este caso, la solución general es más bien

$$t_i = c_5 (b^k)^i + c_6 i (b^k)^i$$

En términos de $T(n)$, esto equivale a

$$T(n) = c_7 n^k + c_8 n^k \log_b (n / n_0) \tag{4.32}$$

para unas constantes c_7 y c_8 adecuadas. Al sustituir esto en la recurrencia original, nuestra manipulación habitual produce un $c_8 = c$ sorprendentemente sencillo. Por tanto, $cn^k \log_b n$ es el término dominante de la ecuación 4.32 porque al principio de este ejercicio se suponía que c era estrictamente positiva. Como n^k es suave y $T(n)$ es asintéticamente no decreciente, concluimos que $T(n) \in \Theta(n^k \log n)$.

Resumiendo,

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \ell < b^k \\ \Theta(n^k \log n) & \text{si } \ell = b^k \\ \Theta(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases} \tag{4.33}$$

El Problema 4.44 da una generalización de este ejemplo. ❑

Comentario: En el análisis de algoritmos, suele ocurrir que las recurrencias se derivan en forma de *des*igualdades. Por ejemplo, se puede obtener

$$T(n) \leq \ell T(n / b) + cn^k \qquad n > n_0$$

en donde n/n_0 es una potencia exacta de b, en lugar de la ecuación 4.29. ¿Qué se puede decir acerca del comportamiento asintótico de esta recurrencia? Obsérvese en primer lugar que no se dispone de información suficiente para determinar el orden *exacto* de $T(n)$, porque solamente se nos da una cota superior de su valor. (Por lo que nosotros sabemos, podría ser $T(n) = 1$ para todo n.) En este caso, lo mejor que podemos hacer es analizar la recurrencia en términos de la notación O. Para esto, introduciremos una recurrencia auxiliar modelada a partir del original, pero definida en términos de una ecuación (no de una inecuación). En este caso:

$$\hat{T}(n) = \begin{cases} T(n_0) & \text{si } n = n_0 \\ \ell \hat{T}(n / b) + cn^k & \text{si } n / n_0 \text{ es una potencia de } b,\ n > n_0. \end{cases}$$

Esta nueva recurrencia se encuentra en el ámbito del ejemplo 4.7.13, salvo que no tenemos evidencia de que $\hat{T}(n)$ sea asintéticamente no decreciente. Por tanto, la ecuación 4.33 es válida para $\hat{T}(n)$, siempre y cuando utilicemos la notación asintótica condicional para restringir n/n_0 de tal modo que sea una potencia de b. Ahora bien, es fácil probar por inducción matemática que $T(n) \leq \hat{T}(n)$ para todo $n \geq n_0$ tal que n/n_0 sea una potencia de b. Ahora bien, está claro que si

$$f(n) \in \Theta(t(n) \mid P(n))$$

y $g(n) \leq f(n)$ para todo n tal que $P(n)$ sea válido, entonces $g(n) \in O(t(n) \mid P(n))$. Por tanto, nuestra conclusión acerca del comportamiento asintótico condicional de

$\hat{T}(n)$ es válida también para $T(n)$, siempre y cuando sustituyamos la Θ por una O. Finalmente, siempre que sepamos que $T(n)$ es asintéticamente no decreciente, podemos invocar la suavidad de las funciones implicadas para concluir que la ecuación 4.33 es válida incondicionalmente para $T(n)$, siempre y cuando una vez más sustituyamos Θ por O. La solución de la recurrencia es, por tanto:

$$T(n) = \begin{cases} O(n^k) & \text{si } \ell < b^k \\ O(n^k \log n) & \text{si } \ell = b^k \\ O(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases}$$

Estudiaremos otras recurrencias que implican desigualdades en la Sección 4.7.6.

Hasta el momento, los cambios de variable que hemos efectuado han sido todos ellos de naturaleza logarítmica. En algunas ocasiones resultan útiles otros cambios de variable muy distintos. Ilustraremos esto con un ejemplo que proviene del algoritmo de divide y vencerás aplicado a la multiplicación de enteros muy grandes (véase la Sección 7.1).

Ejemplo 4.7.14. Considérese una función asintéticamente no decreciente $T(n)$ tal que

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil) + cn \tag{4.34}$$

para todo n suficientemente grande, donde c es alguna constante real y positiva. Tal como se explicaba en el comentario que sigue al ejemplo anterior, tenemos que conformarnos con analizar la recurrencia en términos de la notación O, y no de la notación Θ.

Sea $n_0 \geq 1$ suficientemente grande para que $T(m) \geq T(n)$ para todo $m \geq n \geq n_0/2$, y para que la ecuación 4.34 sea válida para todo $n > n_0$. Considérese cualquier $n > n_0$. Obsérvese primero que

$$\lfloor n/2 \rfloor \leq \lceil n/2 \rceil < 1 + \lceil n/2 \rceil$$

lo cual implica que

$$T(\lfloor n/2 \rfloor) \leq T(\lceil n/2 \rceil) \leq T(1 + \lceil n/2 \rceil)$$

Por tanto, la ecuación 4.34 da lugar a

$$T(n) \leq 3T(1 + \lceil n/2 \rceil) + cn$$

Ahora hacemos un cambio de variable, introduciendo una nueva función $\hat{T}$ tal que $\hat{T}(n) = T(n + 2)$ para todo n. Considérese una vez más cualquier $n > n_0$.

$$\begin{aligned}\hat{T}(n) = T(n+2) &\le 3T(1 + \lceil (n+2)/2 \rceil) + c(n+2) \\ &\le 3T(2 + \lceil n/2 \rceil) + 2cn \quad \text{(porque } n+2 \le 2n\text{)} \\ &= 3\hat{T}(\lceil n/2 \rceil) + 2cn\end{aligned}$$

En particular,

$$\hat{T}(n) \le 3\hat{T}(n/2) + dn \qquad n > n_0$$

cuando n/n_0 es una potencia de 2, en donde $d = 2c$. Este es un caso especial de la recurrencia que se analizaba en el comentario que sigue al Ejemplo 4.7.13, con $\ell = 3$, $b = 2$ y $k = 1$. Dado que $\ell < b^k$, obtenemos que $\hat{T}(n) \in O(n^{\lg 3})$. Finalmente, utilizaremos por última vez el hecho consistente en que $T(n)$ es eventualmente no decreciente: $T(n) \le T(n+2) = \hat{T}(n)$ para todo n suficientemente grande. Por tanto, toda cota superior asintótica de $\hat{T}(n)$ es aplicable igualmente a $T(n)$, lo cual concluye la demostración de que $T(n) \in O(n^{\lg 3})$. ❑

4.7.5 Transformaciones de intervalo

Cuando se hace un cambio de variable, se transforma el dominio de la recurrencia. En lugar de hacer esto, puede resultar útil transformar el intervalo para obtener una recurrencia que tenga una forma que sepamos resolver. Ambas transformaciones pueden utilizarse a la vez en algunas ocasiones. Daremos un ejemplo de este enfoque.

Ejemplo 4.7.15. Considérese la siguiente recurrencia, que define $T(n)$ cuando n es una potencia de 2:

$$T(n) = \begin{cases} 1/3 & \text{si } n = 1 \\ nT^2(n/2) & \text{en caso contrario} \end{cases}$$

El primer paso es un cambio de variable. Supongamos que t_i denota $T(2^i)$:

$$\begin{aligned}t_i = T(2^i) &= 2^i T^2(2^{i-1}) \\ &= 2^i t_{i-1}^2\end{aligned}$$

A primera vista, ninguna de las técnicas vistas hasta el momento parece ser aplicable a esta recurrencia, puesto que no es lineal, y además el coeficiente 2^i no es constante. Para transformar el intervalo (rango), creamos otra recurrencia más empleando u_i para denotar $\lg t_i$:

$$\begin{aligned}u_i = \lg t_i &= i + 2\lg t_{i-1} \\ &= i + 2u_{i-1}\end{aligned}$$

Ahora, una vez reescrita en la forma

$$u_i - 2u_{i-1} = i,$$

la recurrencia se ajusta a la Ecuación 4.10. El polinomio característico es

$$(x-2)(x-1)^2$$

y por tanto todas las soluciones son de la forma

$$u_i = c_1 2^i + c_2 1^i + c_3 i 1^i.$$

Sustituyendo esta solución en la recurrencia para u_i, se obtiene

$$\begin{aligned} i &= u_i - 2u_{i-1} \\ &= c_1 2^i + c_2 + c_3 i - 2(c_1 2^{i-1} + c_2 + c_3(i-1)) \\ &= (2c_3 - c_2) - c_3 i \end{aligned}$$

y por tanto $c_3 = -1$ y $c_2 = 2c_3 = -2$. En consecuencia, la solución general para u_i, si no se tiene en cuenta la condición inicial, es $u_i = c_1 2^i - i - 2$. Esto nos da la solución general para t_i y para $T(n)$.

$$t_i = 2^{u_i} = 2^{c_1 2^i - i - 2}$$

$$T(n) = t_{\lg n} = 2^{c_1 n - \lg n - 2} = \frac{2^{c_1 n}}{4n}$$

Utilizamos la condición inicial $T(1) = 1/3$ para determinar c_1: $T(1) = 2^{c_1} / 4 = 1/3$ implica que $c_1 = \lg(4/3) = 2 - \lg 3$. La solución final es, por tanto:

$$T(n) = \frac{2^{2n}}{4n3^n}.$$

❑

4.7.6 Recurrencias asintóticas

Cuando surgen recurrencias en el análisis de algoritmos, es frecuente que no sean tan "elegantes" como

$$S(n) = \begin{cases} a & \text{si } n = 1 \\ 4S(n \div 2) + bn & \text{si } n > 1 \end{cases} \qquad (4.35)$$

para unas constantes reales y positivas a y b específicas. En su lugar, lo normal es tener que enfrentarse con algo menos preciso, tal como

$$T(n) = 4T(n \div 2) + f(n) \qquad (4.36)$$

cuando n es suficientemente grande, y en donde lo único que sabemos de $f(n)$ está en el orden exacto de n, y no sabemos nada acerca de la condición inicial que define $T(n)$ salvo que es positiva para todo n. Este tipo de ecuación se denomina *recurrencia asintótica*. Afortunadamente, la solución asintótica de una recurrencia tal como la ecuación 4.36 es prácticamente siempre idéntica a la forma de la ecuación 4.35, más sencilla. La técnica general para resolver una recurrencia asintótica consiste en "emparedar" la función que define entre dos recurrencias del tipo más sencillo. Cuando las dos recurrencias más sencillas tienen la misma solución asintótica, la recurrencia asintótica debe de tener también la misma solución. Ilustraremos esto con nuestro ejemplo.

Para las constantes reales y positivas a y b, se define la recurrencia

$$T_{a,b}(n) = \begin{cases} a & \text{si } n = 1 \\ 4T_{a,b}(n \div 2) + bn & \text{si } n > 1 \end{cases}$$

Las técnicas que ya nos resultan familiares se aplican para dar

$$T_{a,b}(n) = (a+b)n^2 - bn$$

siempre y cuando n sea una potencia de 2. Dado que n es una función suave y que $T_{a,b}(n)$ es no decreciente (fácil de probar por inducción matemática), se sigue que $T_{a,b}(n) \in \Theta(n^2)$ independientemente de los valores (positivos) de a y b. De hecho, no hay necesidad de resolver explícitamente la recurrencia porque el resultado del Ejemplo 4.7.13 se aplica directamente una vez que se ha establecido que $T_{a,b}(n)$ es no decreciente.

Para alcanzar nuestra meta, basta demostrar la existencia de cuatro constantes reales y positivas r, s, u y v tales que

$$T_{r,s}(n) \le T(n) \le T_{u,v}(n) \qquad (4.37$$

para todo $n \ge 1$. Dado que tanto $T_{r,s}(n)$ como $T_{u,v}(n)$ están en el orden exacto de n^2 se seguirá que también $T(n) \in \Theta(n^2)$. Un beneficio interesante de este enfoque es que $T_{a,b}(n)$ es no decreciente para cualesquiera valores positivos fijados para a y b lo cual hace posible aplicar la regla de suavidad. Por otra parte, esta regla no se podría haber invocado para simplificar directamente el análisis de $T(n)$ restringiendo nuestra atención al caso en el cual n es una potencia de 2, porque la ecuación 4.36 no ofrece información suficiente para asegurar que $T(n)$ sea no decreciente.

Todavía queda por probar la existencia de r, s, u y v. Para esto, obsérvese que $T_{a,a}(n) = aT_{1,1}(n)$ para todo a, y que $T_{a,b}(n) \le T_{a',b'}(n)$ cuando $a \le a'$ y $b \le b'$ (otras dos demostraciones sencillas por inducción matemática). Sean c y d constantes re-

ales y positivas, y sea n_0 un entero suficientemente grande para que $cn \le f(n) \le dn$ y además

$$T(n) = 4T(n \div 2) + f(n)$$

para todo $n > n_0$. Seleccionamos $r = \min(T(n) / T_{1,1}(n) | 1 \le n \le n_0)$, $s = \min(r, c)$, y $v = \max(u,d)$. Por definición,

$$T(n) \ge rT_{1,1}(n) = T_{r,r}(n) \ge T_{r,s}(n)$$

y además

$$T(n) \le uT_{1,1}(n) = T_{u,u}(n) \le T_{u,v}(n)$$

para todo $n \le n_0$. Esta ecuación constituye la base de la demostración de que la ecuación 4.37 es válida, por inducción matemática. Para el paso de inducción considérese un $n > n_0$ arbitrario y supongamos por la hipótesis de inducción que $T_{r,s}(m) \le T(m) \le T_{u,v}(m)$ para todo $m < n$. Entonces:

$$\begin{aligned} T(n) &= 4T(n \div 2) + f(n) \\ &\le 4T(n \div 2) + dn \\ &\le 4T_{u,v}(n \div 2) + dn \quad \text{por hipótesis de inducción} \\ &\le 4T_{u,v}(n \div 2) + vn \quad \text{puesto que } d \le v \\ &\le T_{u,v}(n) \end{aligned}$$

La demostración de que $T_{r,s}(n) \le T(n)$, que es parecida, completa el argumento.

Ejemplo 4.7.16. La clase importante de recurrencias que se resuelve en el ejemplo 4.7.13 se puede generalizar de una forma similar. Considérese una función $T: \mathbb{N} \to \mathbb{R}^+$ tal que

$$T(n) = \ell T(n \div b) + f(n)$$

para todo n suficientemente grande, en donde $\ell \ge 1$ y $b \ge 2$ son constantes, y $f(n) \in \Theta(n^k)$ para algún $k \ge 0$. Para las constantes arbitrarias reales y positivas x e y se define

$$T_{x,y}(n) = \begin{cases} x & \text{si } n = 1 \\ \ell T_{x,y}(n \div b) + yn^k & \text{si } n > 1 \end{cases}.$$

Resulta fácil mostrar por inducción matemática que $T_{x,y}(n)$ es no decreciente. Por tanto, el ejemplo 4.7.13 es aplicable a su análisis con el mismo resultado en

notación Θ, independientemente del valor de x e y. Por último, podemos mostrar con el enfoque empleado más arriba que nuestra función original $T(n)$ tiene que estar emparedada entre $T_{r,s}(n)$ y $T_{u,v}(n)$ para una elección adecuada de r, s, u y v. La conclusión es que

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \ell < b^k \\ \Theta(n^k \log n) & \text{si } \ell = b^k \\ \Theta(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases}$$

exactamente igual que en el caso de la recurrencia más específica que se daba en el ejemplo 4.7.13. Además, si lo único que sabemos acerca de $f(n)$ es que está en el orden de n^k, en lugar de conocer su orden exacto, seguimos teniendo derecho a alcanzar la misma conclusión en lo concerniente a $T(n)$, salvo que hay que sustituir la Θ por una O en todas partes. (Esta situación es sutilmente distinta de la que se investigaba en el comentario que sigue al ejemplo 4.7.13.) ⌟

4.8 Problemas

Problema 4.1 ¿Cuánto tiempo requiere el siguiente "algoritmo" como función de n?:

```
ℓ ← 0
para i ← 1 hasta n hacer
    para j ← 1 hasta n² hacer
        para k ← 1 hasta n³ hacer
            ℓ ← ℓ + 1
```

Exprese su respuesta en la notación Θ, de la forma más sencilla posible. Puede considerar que cada instrucción individual (incluyendo el bucle de control) es elemental.

Problema 4.2 ¿Cuánto tiempo requiere el siguiente "algoritmo" como función de n?:

```
ℓ ← 0
para i ← 1 hasta n hacer
    para j ← 1 hasta i hacer
        para k ← j hasta n hacer
            ℓ ← ℓ + 1
```

Exprese su respuesta en la notación Θ, de la forma más sencilla posible. Puede considerar que cada instrucción individual (incluyendo el bucle de control) es elemental.

Problema 4.3. Considere el bucle
para $i \leftarrow 1$ **hasta** m **hacer** P

que forma parte de un algoritmo más extenso, el cual trabaja sobre un ejemplar de tamaño n. Sea t el tiempo necesario para cada ejecución de P, que supondremos independiente de i para este problema (pero t podría depender de n). Demostrar que este bucle requiere un tiempo que se encuentra en $\Theta(mt)$ siempre y cuando t esté acotado inferiormente por alguna constante, y siempre y cuando exista un umbral n_0 tal que $m \geq 1$ siempre que $n \geq n_0$. (Recuerde que vimos en la Sección 4.2.2 que la conclusión deseada no sería válida sin estas restricciones.)

Problema 4.4. Demostrar por inducción matemática que los valores de i y j al final de

la k-ésima iteración de Fibiter en la Sección 4.2.2 son f_{k-1} y f_k, respectivamente, en donde f_n denota el n-ésimo número de Fibonacci.

Problema 4.5. Considere el siguiente algoritmo para calcular coeficientes binomiales:

```
función C(n,k)
  si k = 0 o k = n entonces devolver 1
  sino, devolver C(n1, k1) + C(n–1, k)
```

Analizar el tiempo requerido por este algoritmo haciendo la suposición (poco razonable) consistente en que la suma $C(n–1, k–1) + C(n–1, k)$ se puede efectuar en un tiempo constante una vez que tanto $C(n–1, k–1)$ como $C(n–1, k)$ se hayan obtenido recursivamente. Sea $t(n)$ el peor tiempo posible que puede requerir una llamada a $C(n, k)$ para todos los posibles valores de k, con $0 \leq k \leq n$. Expresar $t(n)$ de la forma más sencilla posible en la notación Θ.

Problema 4.6 Considérese el siguiente "algoritmo".

```
procedimiento DC(n)
  si n ≤ 1 entonces terminar
  para i ← 1 hasta 8 hacer DC(n÷2)
  para i ← 1 hasta n3 hacer nuda ← 0
```

Escriba una ecuación asintótica de recurrencia que de el tiempo $T(n)$ requerido por una llamada a $DC(n)$. Utilice el resultado del Ejemplo 4.7.16 para determinar el orden exacto de $T(n)$ en la forma más sencilla posible. *No* reinvente la rueda aquí: aplique el Ejemplo 4.7.16 directamente. ¡La solución completa de este problema no debería ocupar más de 2 o 3 líneas!

Nota: así de sencillo es analizar el tiempo de ejecución de la mayoría de los algoritmos de divide y vencerás; véase el Capítulo 7.

Problema 4.7. Rehacer el Problema 4.6 si la constante 8 que aparece en la línea intermedia del algoritmo *DC* se sustituye por un 9.

Problema 4.8. Rehacer el Problema 4.6 si la constante 8 que aparece en la línea intermedia del algoritmo *DC* se sustituye por un 7.

Problema 4.9. Considérese el siguiente "algoritmo"!

```
procedimiento desperdiciar(n)
    para i ← 1 hasta n hacer
        para j ← 1 hasta i hacer
            escribir i, j, n
  si n > 0 entonces
    para i ← 1 hasta 4 hacer desperdiciar
                                (n÷2)
```

Sea $T(n)$ el número de líneas de salida generadas por una llamada a *desperdiciar*(n). Proporcionar una ecuación de recurrencia para $T(n)$, y utilizar el ejemplo 4.7.16 para determinar el orden exacto de $T(n)$ en la forma más sencilla posible. (*No* le pedimos que resuelva exactamente la recurrencia.)

Problema 4.10. Demostrar por inducción matemática que si $d_0 = n$ y $d_i \leq d_{i-1}/2$ para todo $i \geq 1$, entonces $d_i \leq n/2^i$ para todo $i \geq 0$. (Esto es relevante para el análisis del tiempo requerido para una búsqueda binaria; véase la Sección 4.2.4.)

Problema 4.11. Considérese la siguiente recurrencia para $n \geq 1$:

$$\hat{t}(n) = \begin{cases} c & \text{si } n = 1 \\ \hat{t}(n \div 2) + b & \text{en caso contrario} \end{cases}$$

Utilice la técnica de la ecuación característica para resolver esta recurrencia cuando n sea una potencia de 2. Demostrar por inducción

matemática que $\hat{t}(n)$ es una función asintéticamente no decreciente. Utilice la regla de suavidad para mostrar que $\hat{t}(n) \in \Theta(\log n)$. Por último, concluya que $t(n) \in O(\log n)$, en donde $t(n)$ está dado por la ecuación 4.2. ¿Se puede concluir, a partir de la ecuación 4.2, que $t(n) \in \Theta(\log n)$? ¿Por qué si ó porqué no?

Problema 4.12. Demostrar que la fase de inicialización en la ordenación por casillas (Sección 2.7.2) requiere un tiempo en $\Theta(n + s)$.

Problema 4.13. Vimos en la Sección 4.2.4 que la búsqueda binaria puede hallar un objeto dentro de un vector ordenado de tamaño n en un tiempo que está en $O(\log n)$. Demostrar que en el caso peor se requiere un tiempo que está en $\Omega(\log n)$. Por otra parte, ¿cuál es el tiempo en el caso mejor?.

Problema 4.14. ¿Cuánto tiempo necesita una ordenación por inserción para ordenar n elementos distintos en el caso *mejor*? Indique su respuesta en notación asintótica.

Problema 4.15. Se vio en la Sección 4.4 que para el análisis del caso peor en la ordenación por inserción existe un buen barómetro, que es el número de veces que se comprueba la condición en el bucle **mientras**. Mostrar que este barómetro también resulta adecuado si lo que nos concierne es el comportamiento del algoritmo en el caso peor (véase el problema 4.14).

Problema 4.16. Demostrar que el algoritmo de Euclides tiene su peor rendimiento para entradas de cualquier tamaño dado al calcular el máximo común divisor de dos números consecutivos de la serie de Fibonacci.

Problema 4.17. Dar un algoritmo no recursivo para resolver el problema de las Torres de Hanoi (véase la Sección 4.4). ¡Se considera hacer trampa limitarse a reescribir el algoritmo recursivo empleando una pila explícita para simular las llamadas recursivas!

Problema 4.18. Demostrar que el problema de las Torres de Hanoi con n anillos no se puede resolver con menos de 2^n–1 movimientos de los anillos.

Problema 4.19. Dar un procedimiento similar al algoritmo *contar* de la Sección 4.6 para incrementar un contador binario de m bits. Esta vez, sin embargo, el contador debe quedar lleno de unos en lugar de volver a cero cuando se produzca un desbordamiento. En otras palabras, si el valor actual representado por el contador es v, entonces el nuevo valor después de una llamada a su algoritmo debe de ser mín(v + 1, 2^m–1). Dar el análisis amortizado de su algoritmo. Debe requerir un tiempo amortizado constante para cada llamada.

Problema 4.20. Demostrar la ecuación 4.6 de la Sección 4.7.1 por inducción matemática cuando n es una potencia de 2. Demostrar también por inducción matemática que la función $T(n)$ que se define en la ecuación 4.4 es no decreciente (para todo n, no solamente cuando n es una potencia de 2).

Problema 4.21. Considérense dos constantes reales y positivas a y b. Utilizar suposiciones inteligentes para resolver la recurrencia siguiente cuando $n \geq 1$.

$$t(n) = \begin{cases} a & \text{si } n = 0 \\ nt(n-1) + bn & \text{en caso contrario} \end{cases}$$

Se admite un término de la forma $\sum_{i=1}^{n} 1/i!$ en la solución. Demuestre su respuesta por inducción matemática. Exprese $t(n)$ en notación Θ en la forma más sencilla posible. ¿Cuál es el valor del $\lim_{n\to\infty} t(n) / n!$ en función de a y b? (Nota: $\sum_{i=1}^{\infty} 1/i! = e - 1$, en donde e = 2.7182818... es la base de los logarit-

mos naturales.) Aunque hemos determinado el comportamiento asintótico de esta recurrencia en el problema 1.31, empleando una inducción constructiva, obsérvese que esta vez hemos obtenido una fórmula más precisa para $t(n)$. En particular, hemos obtenido el límite de $t(n)/n!$ cuando n tiende a infinito. Recuerde que este problema es relevante para el análisis del algoritmo recursivo para calcular determinantes.

Problema 4.22. Resolver la recurrencia del ejemplo 4.7.2 por suposiciones inteligentes. ¡Resista la tentación de "hacer trampa" mirando la solución antes de resolver el problema!

Problema 4.23. Demostrar que la ecuación 4.7 (Sección 4.7.2) sólo tiene soluciones de la forma

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

siempre y cuando las raíces $r_1, r_2, \ldots, r_k$ del polinomio característico sean distintas.

Problema 4.24. Completar la solución del ejemplo 4.7.7 determinando el valor de c_1 en función de t_0.

Problema 4.25. Completar la solución del ejemplo 4.7.11 determinando el valor de c_1 en función de t_0.

Problema 4.26. Completar la solución del ejemplo 4.7.12 determinando el valor de c_1 en función de t_0.

Problema 4.27. Completar la solución del Ejemplo 4.7.13 demostrando que $c_8 = c$.

Problema 4.28. Completar el comentario que sigue al ejemplo 4.7.13, demostrando por inducción matemática que $T(n) \leq \hat{T}(n)$ para todo $n \geq n_0$ tal que n/n_0 sea una potencia de b.

Problema 4.29. Resolver exactamente la siguiente recurrencia exáctamente:

$$t_n = \begin{cases} n & \text{si } n = 0 \text{ o } n = 1 \\ 5t_{n-1} - 6t_{n-2} & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.30. Resolver exactamente la siguiente recurrencia.

$$t_n = \begin{cases} 9n^2 - 15n + 106 & \text{si } n = 0, 1 \text{ o } 2 \\ t_{n-1} + 2t_{n-2} - 2t_{n-3} & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.31. Considere la siguiente recurrencia.

$$t_n = \begin{cases} n & \text{si } n = 0 \text{ o } n = 1 \\ 2t_{n-1} - 2t_{n-2} & \text{en caso contrario} \end{cases}$$

Demostrar que es $t_n = 2^{n/2} \sin(n\pi/4)$, no por inducción matemática sino utilizando la técnica de la ecuación característica.

Problema 4.32. Resolver exactamente la siguiente recurrencia.

$$t_n = \begin{cases} n & \text{si } n = 0, 1, 2 \text{ o } 3 \\ t_{n-1} - t_{n-3} - t_{n-4} & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.33. Resolver exactamente la siguiente recurrencia.

$$t_n = \begin{cases} n+1 & \text{si } n = 0 \text{ o } n = 1 \\ 3t_{n-1} - 2t_{n-2} + 3x2^{n-2} & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.34. Resolver exactamente la siguiente recurrencia.

$$T(n) = \begin{cases} a & \text{si } n = 0 \text{ o } n = 1 \\ T(n-1) + T(n-2) + c & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ y la razón áurea $\phi = (1+\sqrt{5})/2$. Obsérvese que esta es la recurrencia 4.1 de la Sección 4.2.3 si $h(n) = c$, lo cual representa el tiempo invertido por una llamada a *Fibrec(n)* si no se cuentan las sumas con coste unitario.

Problema 4.35. Resolver exactamente la siguiente recurrencia:

$$T(n) = \begin{cases} a & \text{si } n = 0 \text{ o } n = 1 \\ T(n-1) + T(n-2) + cn & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ y la razón áurea $\phi = (1+\sqrt{5})/2$. Obsérvese que esto es la Recurrencia 4.1 de la Sección 4.2.3 si $h(n) = cn$, lo cual representa el tiempo invertido por una llamada a *Fibrec(n)* si no se cuentan las sumas con coste unitario. Compare su respuesta con la del Problema 4.34, en el cual se contaban las sumas con coste unitario.

Problema 4.36. Resolver exactamente la siguiente recurrencia, siendo n una potencia de 2.

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 4T(n/2) + n & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.37. Resolver exactamente la siguiente recurrencia, siendo n una potencia de 2:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + \lg n & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.38. Resolver exactamente la siguiente recurrencia, siendo n una potencia de 2

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 5T(n/2) + (n \lg n)^2 & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.39. Resolver exactamente la siguiente recurrencia, siendo n de la forma 2^{2^k}

$$T(n) = \begin{cases} 1 & \text{si } n = 2 \\ 2T(\sqrt{n}) & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.40. Resolver exactamente la siguiente recurrencia

$$T(n) = \begin{cases} n & \text{si } n = 0 \text{ o } n = 1 \\ \frac{1}{\sqrt{2}}T^2(n-1) + \frac{1}{2}T^2(n-2) + n & \text{en caso contrario} \end{cases}$$

Exprese su respuesta del modo más sencillo posible empleando la notación Θ.

Problema 4.41. Resolver exactamente la siguiente recurrencia, siendo n una potencia de 2.

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3/2 & \text{si } n = 2 \\ \frac{3}{2}T(n/2) - \frac{1}{2}T(n/4) - 1/n & \text{en caso contario} \end{cases}$$

Problema 4.42. Resolver exactamente la siguiente recurrencia.

$$t_n = \begin{cases} 0 & \text{si } n = 1 \\ 1/(4 - t_{n-1}) & \text{en caso contrario} \end{cases}$$

Problema 4.43. Resolver exactamente la siguiente recurrencia, en función de las condiciones iniciales a y b.

$$T(n) = \begin{cases} a & \text{si } n = 0 \\ b & \text{si } n = 1 \\ (1 + T(n-1))/T(n-2) & \text{en caso contrario} \end{cases}$$

Problema 4.44. Vimos en el ejemplo 4.7.13 la solución de una recurrencia importante, que es especialmente útil para el análisis de algoritmos tipo divide y vencerás. En algunos casos, sin embargo, se precisa un resultado más general, y la técnica de la ecuación característica no siempre es aplicable. Sean $n_0 \geq 1$ y $b \geq 2$ dos enteros, y sean a y b constantes reales y positivas. Definimos:

$$X = \{n \in \mathbb{N} \mid (\exists i \in \mathbb{N})\ [n = n_0 b^i]\}$$

Sea $f : X \to \mathbb{R}^{\geq 0}$ una función arbitraria. Se define la función $T : X \to \mathbb{R}^{\geq 0}$ mediante la recurrencia

$$T(n) = \begin{cases} d & \text{si } n = n_0 \\ aT(n/b) + f(n) & \text{si } n \in X, n > n_0 \end{cases}.$$

Sea $p = \log_b a$. Resulta que la forma más sencilla de expresar $T(n)$ en notación asintótica depende de la relación entre $f(n)$ y n^p. En lo que sigue, toda la notación asintótica es implícitamente condicional con respecto a que $n \in X$. Demostrar que

1. Si se hace $f(n_0) = d$, lo cual no tiene importancia a efectos de la definición de T, entonces el valor de $T(n)$ está dado por una simple suma cuando $n \in X$:

$$T(n) = \sum_{i=0}^{\log_b (n/n_0)} a^i f(n/b^i).$$

2. Sea ε cualquier constante real estrictamente positiva; entonces

$$T(n) \in \begin{cases} \Theta(n^p) & \text{si } f(n) \in O(n^p/(\log n)^{1+\varepsilon}) \\ \Theta(f(n) \log n \log\log n) & \text{si } f(n) \in \Theta(n^p/\log n) \\ \Theta(f(n) \log n) & \text{si } f(n) \in \Theta(n^p/(\log n)^{1+\varepsilon}) \\ \Theta(f(n)) & \text{si } f(n) \in \Theta(n^{p-\varepsilon}). \end{cases}$$

La tercera alternativa incluye a $f(n) \in \Theta(n^p)$ al seleccionar $\varepsilon = 1$.

3. Como caso especial de la primera alternativa, $T(n) \in \Theta(n^p)$ siempre que $f(n) \in O(n^r)$ para alguna constante real positiva $r < p$.

4. La primera alternativa se puede generalizar para admitir casos tales como

$$f(n) \in O(n^p / ((\log n)(\log\log n)^{1+\varepsilon}));$$

también se obtiene $T(n) \in \Theta(n^p)$ si $f(n) \in O(n^p g(n))$, en donde $g(n)$ es no creciente y $\sum_{n \in X} g(n)$ converge.

5. La última alternativa se puede generalizar para incluir casos tales como

$$f(n) \in \Theta\left(n^{p+\varepsilon} \log n\right) \text{ o bien } f(n) \in \Theta\left(n^{p+\varepsilon}/\log n\right);$$

también se obtiene

$$T(n) \in \Theta\left(f(n)\right) \text{ si } f(n) \in \Omega\left(n^{p+\varepsilon}\right)$$

y si $f(n) \geq akf(n/b)$ para alguna constante $k > 1$ y para todos los $n \in X$ suficientemente grandes.

4.9 REFERENCIAS Y TEXTOS MÁS AVANZADOS

El nombre Torres de Hanoi corresponde a un juguete inventado por el matemático francés Édouard Lucas en 1883: véase Gardner (1959). Buneman y Levy (1980) y Dewdney (1984) dan soluciones para el problema 4.17.

El análisis amortizado ha sido popularizado por Tarjan (1985).

Los principales aspectos matemáticos del análisis de algoritmos se pueden hallar en Greene y Knuth (1981). En Purdom y Brown (1985) y Rawlins (1992) se hallará una buena cantidad de técnicas para el análisis de algoritmos.

Se explican varias técnicas para resolver recurrencias, incluyendo la ecuación característica y el cambio de variable, en Lueker (1980). Para un tratamiento matemático más rigurosos, véase Knuth (1968) o bien Purdom y Brown (1985). El artículo de Bentley, Haken y Saxe (1980) es especialmente relevante para las recurrencias que surgen del análisis de algoritmos de divide y vencerás (véase el Capítulo 7). El problema 4.44 procede parcialmente de Verma (1994), que presenta otros resultados generales.

Capítulo 5

Estructuras de datos

El uso de unas estructuras de datos bien escogidas suele ser un factor crucial en el diseño de algoritmos eficientes. Sin embargo, este libro no pretende ser un manual de estructuras de datos. Suponemos que el lector ya posee buenos conocimientos prácticos de nociones tales como las matrices, los registros y los distintos tipos de datos estructurados que se obtienen empleando punteros. También suponemos que los conceptos matemáticos de grafos dirigidos y no dirigidos le resultan razonablemente familiares, y que el lector conoce la forma de representar eficientemente estos objetos en una computadora.

El capítulo comienza con una breve revisión de los aspectos más importantes de estas estructuras elementales de datos. La revisión incluye un resumen de sus propiedades esenciales desde el punto de vista de la algoritmia. Por esta razón, también los lectores que conozcan bien el material básico harían bien en estudiar brevemente las primeras secciones. Las tres últimas secciones del capítulo presentan las nociones menos elementales de *montículos (heaps)* y *particiones* (*conjuntos disjuntos*). Estas estructuras, que se han elegido porque se emplearán en capítulos subsiguientes, ofrecen también ejemplos interesantes en el análisis de algoritmos. Es probable que casi todos los lectores necesiten leer con bastante detalle las secciones que conciernen a estas estructuras de datos menos familiares.

5.1 MATRICES (ARRAYS)[1], PILAS Y COLAS

Una *matriz* es una estructura de datos que consta de un número fijo de ítems del mismo tipo. (Hablaremos de *ítems* o de *elementos* indistintamente.) En una máquina, estos elementos suelen almacenarse en posiciones contiguas de almacenamiento. En una matriz monodimensional, el acceso a todo ítem particular se efectúa especificando un solo *subíndice* o *índice*. Por ejemplo, podríamos declarar una matriz monodimensional de enteros en la forma siguiente:

tab: **matriz**[1..50] **de** *enteros*

Aquí, *tab* es una matriz de 50 enteros indexados (*indizados*) desde 1 hasta 50; *tab*[1] se refiere al primer elemento de la matriz, y *tab*[50] alude al último. Es natural pensar que los elementos están dispuestos de izquierda a derecha, así que podemos también refererirnos a *tab*[1] como el elemento de la izquierda, y así sucesivamente. Con frecuencia, dejaremos sin especificar el tipo de los elementos de la matriz cuando el contexto lo haga evidente.

Desde el punto de vista que nos interesa en este libro, la propiedad esencial de una matriz es que podemos calcular la dirección de cualquier elemento dado en un tiempo constante. Por ejemplo, si sabemos que la matriz anterior *tab* comienza en la dirección 5000, y que las variables enteras ocupan 4 *bytes* de almacenamiento cada una, entonces la dirección del elemento cuyo índice es k está dada claramente por $\mathbf{4996 + 4k}$. Aun cuando pensemos que merece la pena comprobar que k se encuentra realmente entre 1 y 50, el tiempo necesario para calcular la dirección sigue estando acotado por una constante. Se sigue que el tiempo necesario para leer el valor de un solo elemento, o para cambiar ese valor, se encuentra en $O(1)$; en otras palabras, podemos tratar estas operaciones como si fuesen elementales.

Por otra parte, toda operación que implique a todos los elementos de una matriz tenderá a requerir más tiempo a medida que crezca el tamaño de la matriz. Supongamos que nos estamos enfrentando a una matriz de tamaño variable n; esto es, la matriz consta de n elementos. Entonces una operación tal como dar valor inicial a todos los elementos, o buscar el mayor elemento, va a requerir normalmente un tiempo que será proporcional al número de elementos que haya que examinar. En otras palabras, esas operaciones requieren un tiempo que está en $\Theta(n)$.

Las matrices monodimensionales permiten implementar eficientemente la estructura de datos llamada *pila*. Aquí los elementos se añaden a la estructura y después se sacan de ella, de tal forma que el último en llegar es el primero en salir [2]. La situación se puede representar utilizando una matriz llamada *pila*, digamos, cuyo índice vaya desde uno hasta el tamaño máximo que requiera la pila, y cuyos elementos sean del tipo correcto, junto con un contador. Para dejar vacía la pila, se da al contador el valor cero; para añadir un elemento a la pila, se incrementa el contador, y entonces se escribe en *pila*[*contador*] ese elemento; para eliminar un elemento, se lee el valor de *pila*[*contador*] y se decrementa el contador. Se pueden añadir comprobaciones para asegurar que no se pongan en la pila más elementos que los reservados, y que no se eliminen elementos de una pila vacía. La adición de elementos a una pila suele denominarse operación **apilar** *push*, mientras que la eliminación de elementos se denomina **desapilar** *pop*.

La estructura de datos denominada *cola* también se puede implementar de forma bastante eficiente en una matriz monodimensional. Aquí los elementos se añaden y se eliminan de tal manera que el primero en entrar es el primero en salir, véase el problema 5.2. La adición de un elemento se denomina una operación de **poner** *"enqueue"*, mientras que la eliminación de un elemento se llama **quitar** *"dequeue"*[3]. Tanto para pilas como para colas, una desventaja del uso de matrices para la implementación es que normalmente hay que reservar espacio al principio para el máximo número de elementos que se prevea; si en alguna ocasión el es-

pacio no resultara suficiente, es difícil reservar más, mientras que si se reserva demasiado espacio, es un desperdicio.

Los elementos de una matriz pueden ser de cualquier tipo de longitud fija: esto es para que la dirección de cualquier elemento se pueda calcular con facilidad. El índice es casi siempre un entero. Sin embargo, otros tipos de los denominados «escalares» se pueden emplear también. Por ejemplo:

lettab: **matriz**['a'..'z'] **de** valor

es una posible forma de declarar 26 valores, indexados mediante las letras de la 'a' hasta la 'z'. No se permite indexar una matriz empleando números reales, ni tampoco se permite indexar una matriz empleando estructuras tales como las cadenas o los conjuntos. Si se permite esta posibilidad, entonces ya no se puede considerar como operación elemental el acceso a un elemento de la matriz. Sin embargo, existe una estructura de datos más general, llamada *tabla asociativa*, que se describe en la Sección 5.6 y *sí* permite tales índices.

Los ejemplos dados hasta el momento implican todos ellos estucturas unidimensionales, esto es, matrices a cuyos elementos se accede empleando un solo índice. Se pueden declarar matrices con dos o más índices de forma similar. Por ejemplo:

matriz: **matriz**[1..20, 1..20] **de** *complejo*[1]

es una posible forma de declarar una matriz que contiene 400 elementos del tipo *complejo*. Una referencia a cualquier elemento concreto, tal como *matriz*[5, 7] requiere ahora dos índices. Lo esencial sigue siendo, sin embargo, que es posible calcular la dirección de cualquier elemento dado en un tiempo constante, así que se puede tomar la lectura o modificación de su valor como operación elemental. Evidentemente, si las dos dimensiones de una matriz bidimensional dependen de algún parámetro n, tal como en el caso de una matriz $n \times n$, entonces las operaciones tales como dar valor inicial a todos los elementos de la matriz, o buscar el elemento mayor, requieren ahora un tiempo que está en $\Theta(n^2)$.

Dijimos anteriormente que el tiempo necesario para *dar un valor inicial* a todos los elementos de una lista de tamaño n está en $\Theta(n)$. Supongamos, sin embargo, que no es preciso dar valor inicial a todos los valores, sino que lo único que se precisa saber es si se le ha dado o no valor inicial, y en tal caso obtener su valor. Si estamos dispuestos a emplear bastante más espacio, la técnica denominada *iniciación virtual* nos permite evitar el tiempo que se invierte en dar valor a todas las entradas de la lista. Supongamos que la matriz que hay que iniciar virtualmente es $T[1..n]$. Necesitamos también dos listas auxiliares de enteros del mismo tamaño que T y un contador entero. Sean $a[1..n]$ y $b[1..n]$ estas dos listas auxiliares, y sea *ctr* el contador. Al comenzar, damos simplemente a *ctr* el valor 0, y dejamos las listas a, b y T con aquellos valores que pudieran contener.

En lo sucesivo, *ctr* nos dice cuántos elementos de T han recibido valor inicial, mientras que los valores desde $a[1]$ hasta $a[ctr]$ nos dicen cuáles son esos elemen-

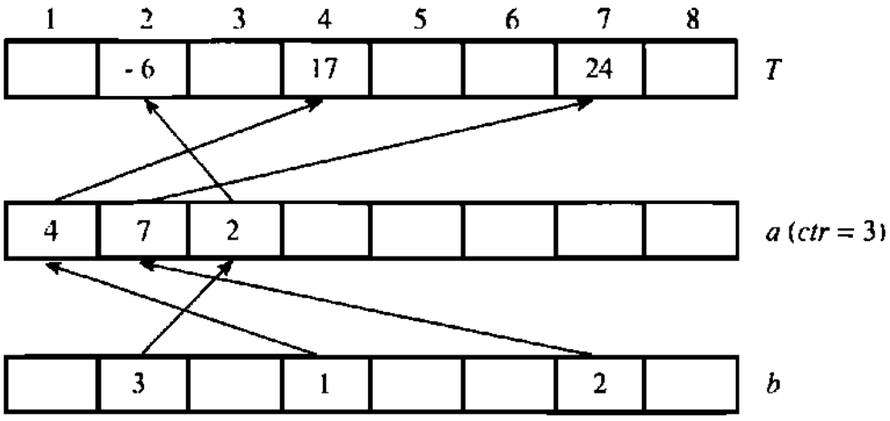


Figura 5.1. Una lista iniciada virtualmente

tos: $a[1]$ señala el elemento que se inicialice en primer lugar, $a[2]$ indica el segundo en ser inicializado, y así sucesivamente; véase la figura 5.1, en la cual se han inicializado tres elementos de la matriz T. Además, si $T[i]$ ha sido el elemento inicializado en k-ésimo lugar, entonces $b[i] = k$. De esta manera, los valores de a apuntan a T y los valores de b apuntan a a, tal como se muestra en la figura.

Para determinar si se ha asignado un valor a $T[i]$, comprobamos primero si $1 \leq b[i] \leq ctr$. Si no se cumple, podemos asegurar que $T[i]$ no ha sido iniciado. Si se cumple, no estamos seguros de si $T[i]$ ha sido inicializado o no: quizá sea un accidente que $b[i]$ tenga un valor plausible. Sin embargo, si *realmente* se ha asignado un valor a.$T[i]$, entonces ha sido el $b[i]$-ésimo elemento en ser iniciado. Podemos comprobar esto estudiando si $a[b[i]] = i$. Dado que los ctr primeros valores de la lista a han sido iniciados con certeza, y que $1 \leq b[i] \leq ctr$, no puede ser un accidente que $a[b[i]] = i$, así que esta comprobación es concluyente: si resulta afirmativa, entonces $T[i]$ ha sido iniciado, y si no, no.

Para asignar un valor a $T[i]$ por primera vez, se incrementa el contador ctr, se da a $a[ctr]$ el valor i, se da a $b[i]$ el valor ctr, y se carga el valor necesario en $T[i]$. En las sucesivas asignaciones a $T[i]$, por supuesto, no es necesario nada de esto. El problema 5.3 le invita a rellenar los detalles. El espacio adicional requerido por esta técnica es un múltiplo constante del tamaño de T.

5.2 REGISTROS Y PUNTEROS (APUNTADORES)[5]

Del mismo modo que una matriz tiene un número fijo de elementos del mismo tipo, un *registro* es una estructura de datos que consta de un número fijo de elementos, que suelen llamarse *campos* en este contexto, y que son de tipos posiblemente distintos. Por ejemplo, si la información acerca de una persona que nos interesa consta de nombre, edad, peso y sexo, quizá necesitemos utilizar una estructura de datos de la forma siguiente:

tipo *persona* = **registro**
nombre : *cadena*
edad : *entero*
peso : *real*
varon : *booleano*

Entonces, si *Juan* es una variable de este tipo, utilizaremos la *notación de punto* para hacer alusión a los campos; por ejemplo, *Juan.nombre* es una cadena y *Juan.edad* es un entero.

Las matrices pueden aparecer como elementos de un registro, y los registros se pueden almacenar en matrices. Por ejemplo, si declaramos

clase: **matriz**[1..50] **de** *persona*

entonces la matriz *clase* contiene 50 registros. Los atributos del séptimo elemento de la clase se denotan en la forma *clase*[7].*nombre*, *clase*[7].*edad* y así sucesivamente. Tal como sucede con las listas, dado que los registros contienen tan sólo elementos de longitud fija, la dirección de todo elemento particular se puede calcular en un tiempo constante, así que las consultas o modificaciones del valor de un campo se pueden considerar como operaciones elementales.

Hay muchos lenguajes de programación que permiten crear y destruir registros dinámicamente. (Algunos lenguajes permiten hacer esto con las matrices, pero son menos frecuentes.) Ésta es una de las razones por las cuales se suelen utilizar los registros en conjunción con los punteros. Una declaración como

tipo *jefe* = ↑*persona*

dice que *jefe* es un puntero de un registro cuyo tipo es *persona*. Es posible crear dinámicamente uno de estos registros mediante una sentencia tal como

jefe ← **crea**[6] *persona*

Ahora *jefe*↑ (obsérvese que aquí la flecha sigue al nombre) significa «el registro al que apunta *jefe*». Para hacer alusión a los campos de este registro, utilizaremos *jefe*↑.*nombre*, *jefe*↑.*edad* y así sucesivamente. Si un puntero tiene el valor especial **nulo** "*nil*", entonces no apunta a ningún registro.

5.3 LISTAS

Una *lista* es una colección de elementos de información dispuestos en un cierto orden. A diferencia de las matrices y registros, el número de elementos de la lista no suele estar fijado, ni suele estar limitado por anticipado. La estructura de datos correspondiente debería de permitirnos determinar, por ejemplo, cuál es el primer

elemento de la estructura, cuál es el último, y cuál es el predecesor y el sucesor (si existen) de cualquier elemento dado. En una máquina, el espacio correspondiente a cualquier elemento de información dado suele denominarse un *nodo*. Además de la información en cuestión, un nodo puede contener uno o más punteros. Estas estructuras suelen representarse gráficamente mediante cuadros y flechas, tal como en la figura 5.2. La información asociada a cada nodo se muestra dentro del cuadro correspondiente, y las flechas muestran los enlaces que van desde el nodo a su sucesor.

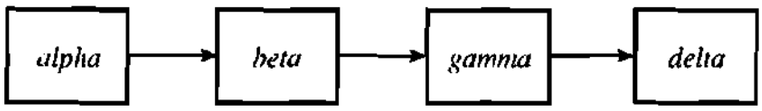


Figura 5.2. Una lista

Las listas admiten un cierto número de operaciones: quizá sea necesario insertar un nodo adicional, borrar un nodo, copiar una lista, contar el número de elementos que contienen, y así sucesivamente. Las distintas implementaciones de computadora que se suelen utilizar difieren en la cantidad de espacio requerido, y en la facilidad para efectuar ciertas operaciones. Aquí nos contentaremos con mencionar las técnicas más conocidas.

Implementada en forma de matriz mediante la declaración

```
tipo lista = registro
       contador: 0..longmax
       valor: matriz[1..longmax] de información
```

los ítems de una lista ocupan las posiciones que van desde *valor*[1] hasta *valor*[*contador*], y el orden de los elementos es el mismo que el orden de sus índices dentro de la matriz. Empleando esta implementación, se puede hallar rápidamente los elementos primero y último de la lista, así como también el predecesor y el sucesor de cualquier elemento dado. Por otra parte, como se vio en la Sección 5.1, insertar un nuevo elemento o borrar uno de los elementos existentes requiere un número de operaciones que, en el caso peor, está en el orden del tamaño actual de la lista. Se indicaba allí, sin embargo, que esta implementación es especialmente eficiente para una importante estructura denominada «pila»; y una pila se puede considerar como una clase de lista en la cual la adición y borrado de elementos se permiten solamente en un extremo indicado de la lista. A pesar de ello, esta implementación de una pila puede presentar la grave desventaja de que todo el almacenamiento precisado potencialmente esté reservado a lo largo de toda la vida del programa.

Por otra parte, si se emplean punteros para implementar una estructura de lista, entonces los nodos suelen ser registros de una forma similar a la siguiente:

```
tipo lista = ↑nodo
tipo nodo = registro
        valor: información
        siguiente: ↑nodo
```

en donde todos los nodos salvo el último incluyen un puntero explícito a su sucesor. El puntero del último nodo tiene el valor especial *nulo*, para indicar que no apunta a ningún nodo. En este caso, siempre que se utilice un lenguaje de programación adecuado, el espacio que se necesita para representar la lista se puede asignar y recuperar dinámicamente a medida que avanza el programa. Por tanto, dos o más listas podrían compartir el mismo espacio; además, no es necesario saber *a priori* la longitud que va a tener ninguna lista en particular.

Aun cuando se utilicen punteros adicionales para asegurar un acceso rápido al primer elemento de la lista, y también al último, cuando se utiliza esta representación resulta difícil examinar el k-ésimo elemento, para un k arbitrario, sin tener que seguir k punteros e incurrir por tanto en un tiempo que está en $O(k)$. Sin embargo, una vez que se ha encontrado un elemento, la inserción de un nuevo nodo o el borrado de un nodo ya existente se puede hacer rápidamente, copiando o cambiando únicamente unos pocos campos. En nuestro ejemplo, se utiliza un solo puntero en cada nodo para indicar su sucesor. Por tanto resulta sencillo recorrer la lista en una dirección, pero no en la otra. Si se puede aceptar un mayor coste de almacenamiento, basta con añadir a cada nodo un nuevo puntero para permitir que la lista se recorra rápidamente en ambas direcciones.

Los libros elementales de programación están plagados de variaciones sobre este tema. Las listas pueden ser circulares, con el último elemento apuntando al primero, o pueden tener una primera celda especial, distinta, que contenga por ejemplo el número de nodos que hay en la lista. Sin embargo, en este libro no utilizaremos esas estructuras.

5.4 GRAFOS

Hablando intuitivamente, un *grafo* es un conjunto de nodos unidos por un conjunto de líneas o flechas. Considérese, por ejemplo, la figura 5.3. Distinguiremos entre grafos dirigidos y grafos no dirigidos. En los grafos dirigidos, los nodos están unidos mediante flechas llamadas *aristas*. En el ejemplo de la figura 5.3, existe una arista que va desde *alfa* hasta *gamma*, y otra desde *gamma* hasta *alfa*. Los nodos *beta* y *delta*, sin embargo, solo están unidos en la dirección indicada. En el caso de un grafo no dirigido, los nodos están unidos mediante líneas sin indicación de dirección, que también se llaman aristas. Tanto en los grafos dirigidos como en los no dirigidos, las secuencias de aristas pueden formar *caminos* y *ciclos*. Se dice que un grafo es *conexo* si se puede llegar desde cualquier nodo hasta cualquier otro siguiendo una secuencia de aristas; en el caso de un grafo dirigido, se permite circular en sentido inverso a lo largo de una flecha. Diremos que un grafo dirigido es *fuertemente conexo* si se puede pasar desde cualquier nodo hasta cual-

quier otro siguiendo una secuencia de aristas, pero respetando esta vez el sentido de las flechas.

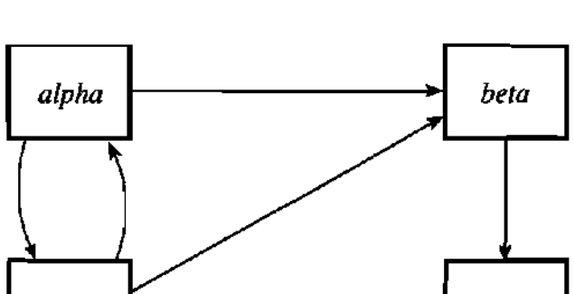


Figura 5.3. Un grafo dirigido

Nunca hay más de dos flechas que unan dos nodos dados de un grafo dirigido, y si hay *dos* flechas, entonces tienen que ir en sentidos opuestos; en un grafo no dirigido nunca hay más de una línea que una dos nodos dados. Formalmente, un grafo es por tanto una pareja $G = \langle N, A \rangle$ en donde N es un conjunto de nodos y A es un conjunto de aristas. Una arista que vaya desde el nodo a hasta el nodo b de un grafo dirigido se denotará mediante el par ordenado (a, b), mientras que una arista que una los nodos a y b en un grafo no dirigido se denota mediante el conjunto $\{a, b\}$. (Recuerde que un conjunto es una colección *desordenada* de elementos; véase la Sección 1.4.2.) Por ejemplo, la figura 5.3 es una representación informal del grafo $G = \langle N, A \rangle$, en donde

$$N = \{alfa, beta, gamma, delta\}$$

$$A = \{(alfa, beta), (alfa, gamma), (beta, delta),$$

$$(gamma, alfa), (gamma, beta), (gamma, delta)\}$$

Hay por lo menos dos maneras evidentes de representar un grafo en una computadora. La primera utiliza una *matriz de adyacencia*:

```
tipo grafoadya = registro
        valor: matriz[1..numnodos] de información
        adyacente: matriz[1..numnodos, 1..numnodos] de Boolean
```

Si el grafo incluye una arista desde el nodo i hasta el nodo j, entonces *adyacente*$[i, j]$ = *verdadero*; en caso contrario *adyacente*$[i, j]$ = *falso*. En el caso de un grafo no dirigido, la matriz será necesariamente simétrica.

Con esta representación, es fácil ver si existe o no una arista entre dos nodos dados: buscar un valor en la matriz requiere un tiempo constante. Por otra parte, si deseamos examinar todos los nodos que estén conectados con algún nodo dado, tenemos que recorrer toda una fila completa de la matriz, en el caso de un grafo no dirigido, o bien tanto una fila completa como una columna completa en el ca-

so de un grafo dirigido. Esto requiere un tiempo que está en $\Theta(numnodos)$, el número de nodos que hay en el grafo, independientemente del número de aristas que entren o salgan de ese nodo particular. El espacio requerido para representar un grafo de esta manera es cuadrático con respecto al número de nodos.

Lo que sigue es una segunda representación posible:

```
tipo grafolista = matriz[1..numnodos] de
                  registro
                  valor: información
                  vecinos: lista
```

Aquí se asocia a cada nodo i una lista formada por sus vecinos, esto es, una lista formada por aquellos nodos j tales que existe una arista de i a j (en el caso de un grafo dirigido) o entre i y j (en el caso de un grafo no dirigido). Si el número de aristas del grafo es pequeño, esta representación utiliza menos espacio que la dada anteriormente. También puede que sea posible en este caso examinar todos los vecinos de un nodo dado en menos de *numnodos* operaciones de análisis en el caso medio. Por otra parte, determinar si existe o no una conexión directa entre dos nodos dados i y j nos obliga a recorrer la lista de vecinos del nodo i (y posiblemente también los del nodo j, en el caso de un grafo dirigido), lo cual es menos eficiente que buscar un valor booleano en una matriz.

5.5 ÁRBOLES

Un *árbol* (hablando con propiedad, un *árbol libre*) es un grafo acíclico, conexo y no dirigido. De manera equivalente, un árbol se puede definir como un grafo no dirigido en el cual existe exactamente un camino entre todo par de nodos dado. Puesto que un árbol es una clase de grafo, las mismas representaciones que se emplean para implementar los grafos se pueden utilizar para representar árboles. La figura 5.4(a) muestra dos árboles, cada uno de los cuales posee cuatro nodos; véase el problema 5.7. Los árboles tienen un cierto número de propiedades sencillas, de las cuales las más importantes son probablemente las siguientes:

- ◊ Un árbol con n nodos posee exactamente n-1 aristas.
- ◊ Si se añade una única arista a un árbol, entonces el grafo resultante contiene un único ciclo.
- ◊ Si se elimina una única arista de un árbol, entonces el grafo resultante ya no es conexo.

En este libro nos interesaremos principalmente por los *árboles con raíz*. Se trata de aquellos árboles en los cuales hay un nodo, llamado *raíz*, que es especial. Cuando se dibuja un árbol con raíz, se acostumbra a dibujar la raíz en la parte superior, tal como en un árbol genealógico, con todas las aristas bajando de él. La figura 5.4(b) ilustra cuatro árboles con raíz diferente, cada uno de los cuales posee cua-

tro nodos. Una vez más se puede verificar fácilmente que éstos son los únicos árboles con raíz de cuatro nodos que existen. Cuando no hay peligro de confusión, utilizaremos el término sencillo «árbol» en lugar de «árbol con raíz», más correcto, puesto que casi todos nuestros ejemplos son de este tipo.

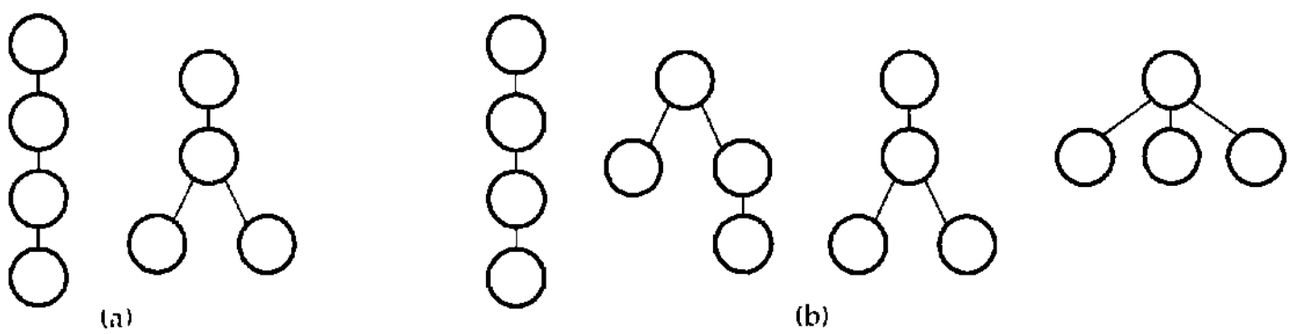


Figura 5.4. (a) árboles, y (b) árboles con raíz de cuatro nodos

Extendiendo la analogía con un árbol genealógico, se acostumbra a utilizar términos tales como «padre» e «hijo» para describir la relación entre nodos adyacentes. Por tanto en la figura 5.5, *alfa*, la raíz del árbol, es el *padre* de *beta* y de *gamma*; *beta* es el padre de *delta*, *épsilon* y *zeta*, y el *hijo* de *alfa*; mientras que *épsilon* y *zeta* son los *hermanos* de *delta*. Un *antepasado* de un nodo es o bien el nodo en sí (lo cual no coincide con la acepción cotidiana) o bien su padre, el padre de su padre, y así sucesivamente. Por tanto, tanto *alfa* como *zeta* son antepasados de *zeta*. Un *descendiente* de un nodo se define de forma análoga, incluyendo una vez más al propio nodo.

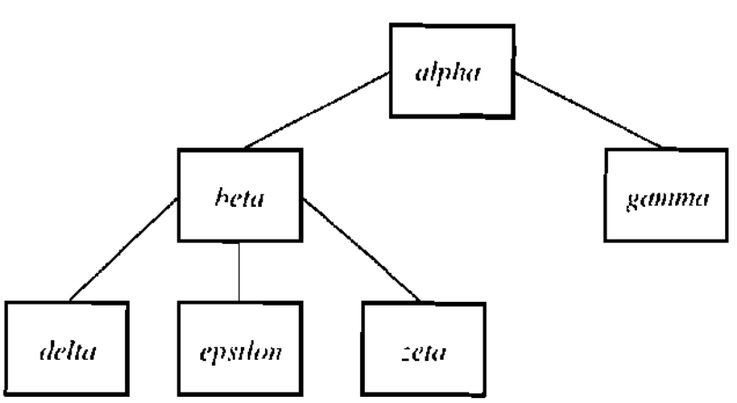


Figura 5.5. Un árbol con raíz

Una *hoja* de un árbol con raíz es un nodo que no tiene hijos; los demás nodos se denominan *nodos internos*. Aun cuando no hay nada en la definición que lo indique, las ramas de los árboles con raíces suelen considerarse ordenadas de izquierda a derecha: en el ejemplo anterior, *beta* está situado a la izquierda de *gamma*, y —una vez más por analogía con un árbol genealógico— se dice que *delta* es el hermano *mayor* de *épsilon* y *zeta*. Los dos árboles de la figura 5.6 pueden considerarse, por tanto, diferentes.

En una computadora, todo árbol con raíz se puede representar empleando nodos del tipo siguiente:

tipo *nodoarbol* 1 = **registro**
valor: *información*
hijo-mayor, hermano-siguiente: ↑*nodoarbol* 1

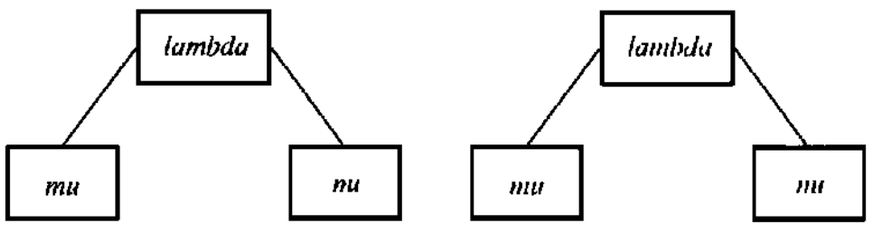


Figura 5.6. Dos árboles con raíces distintas

El árbol con raíz que se muestra en la figura 5.5 se representaría tal como en la figura 5.7, en la cual las flechas muestran la dirección de los punteros empleados en la representación por computadora, y no la dirección de las aristas del árbol (que es, por supuesto, un grafo no dirigido.) Insistimos en que esta representación se puede utilizar para *cualquier* árbol con raíz; tiene la ventaja de que todos los nodos se pueden representar utilizando la misma estructura **registro**, independientemente del número de hijos y hermanos que posean. Sin embargo, hay muchas operaciones que resultan ineficientes cuando se emplea esta representación mínima: no resulta evidente la forma de hallar el padre de un nodo dado, por ejemplo (pero véase el problema 5.10.)

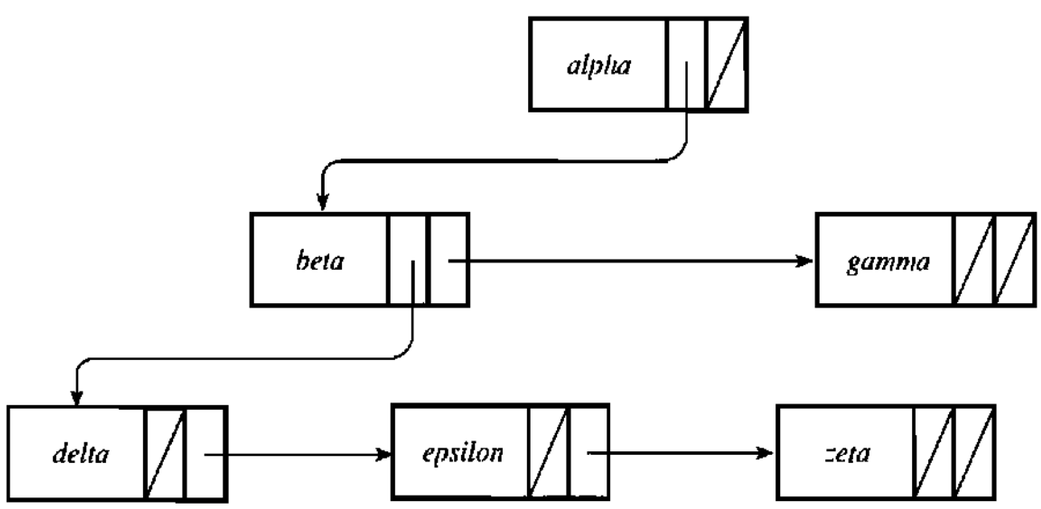


Figura 5.7 Una posible representación por computadora para un árbol con raíz

Otra representación adecuada para *cualquier* árbol con raíz utiliza nodos del tipo

tipo *nodoarbol* 2 = **registro**
valor: *información*
padre: ↑*nodoarbol*2

donde ahora cada nodo contiene un único puntero que lleva a su padre. Esta representación es la más económica en términos de espacio que cabría esperar, pero es poco eficiente a no ser que todas las operaciones del árbol impliquen comenzar en un nodo y subir, sin descender nunca. (Para una aplicación en la cual es esto exactamente lo que se necesita, véase la Sección 5.9.) Además, no se representa el orden de los hermanos.

Normalmente, se puede diseñar una representación adecuada para una cierta aplicación comenzando por una de estas representaciones generales, y añadiendo punteros suplementarios, por ejemplo que apunten al padre o al hermano mayor de un nodo dado. De esta manera, podemos acelerar aquellas operaciones que deseemos efectuar eficientemente incurriendo en el coste de un incremento de espacio.

Con frecuencia, tendremos ocasión de utilizar *árboles binarios*. En tales árboles, cada nodo puede tener 0, 1 o 2 hijos. De hecho, casi siempre supondremos que cada nodo posee dos punteros, uno a su izquierda y otro a su derecha, cualquiera de los cuales puede ser **nulo**. Cuando se hace esto, aunque la metáfora queda un poco forzada, tendemos de forma natural a hablar del hijo izquierdo y el hijo derecho, y la posición ocupada por los hijos resulta significativa: un nodo que tenga un hijo izquierdo pero no un hijo derecho no puede ser nunca igual que un nodo que tenga un hijo derecho pero no un hijo izquierdo. Por ejemplo, los dos árboles binarios de la figura 5.8 no son iguales: en el primer caso, *b* es el hijo izquierdo de *a*, y falta el hijo derecho, mientras que en el segundo caso *b* es el hijo derecho de *a* y falta el hijo izquierdo.

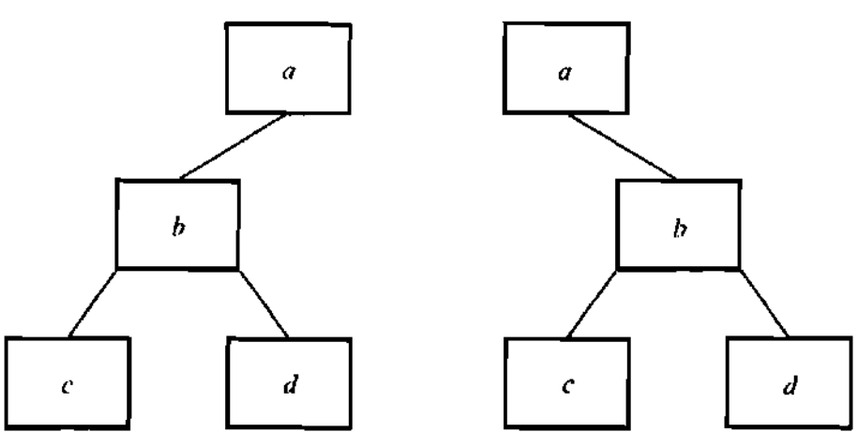


Figura 5.8. Dos árboles binarios diferentes

Si en cada nodo de un árbol con raíz no puede haber más de *k* hijos, decimos que se trata de un *árbol k-ario*. Una representación evidente emplea nodos del tipo siguiente:

```
tipo nodo-k-ario = registro
     valor: información
     hijo: matriz[1..k] de ↑nodo-k-ario
```

En el caso de un árbol binario, también podemos definir

tipo *nodo-binario* = **registro**
valor: *información*
hijo-izquierdo, hijo-derecho: ↑ *nodo-binario*

Hay veces en las cuales también es posible, según se verá en la sección siguiente, representar un árbol *k*-ario empleando una matriz sin ningún puntero explícito.

Un árbol binario es un *árbol de búsqueda* si el valor contenido en todos los nodos internos es mayor o igual que los valores contenidos en su hijo izquierdo o en cualquiera de los descendientes de ese hijo, y menor o igual que los valores contenidos en su hijo derecho o en cualquiera de los descendientes de ese hijo. La figura 5.9 da un ejemplo de un árbol de búsqueda. La figura muestra el valor contenido en cada nodo. Esta estructura es interesante porque, como implica el nombre, permite buscar valores en el árbol de forma eficiente. En el ejemplo, aun cuando el árbol contiene 7 elementos, podríamos hallar el 27, digamos, con sólo 3 comparaciones. La primera, que se hace con el valor 20 almacenado en la raíz, nos dice que el 27 se encuentra en el sub-árbol derecho (si es que está en el árbol); la segunda, que se hace con el 34 almacenado en la raíz del sub-árbol derecho, nos dice que busquemos por la izquierda, y la tercera encuentra el valor que buscamos.

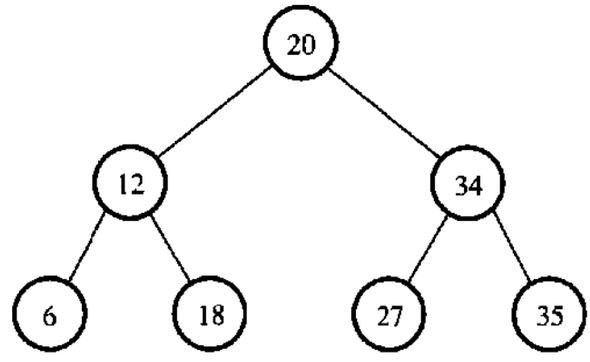


Figura 5.9. Un árbol de búsqueda

El procedimiento de búsqueda esbozado anteriormente se puede describir más formalmente en la forma siguiente:

función *buscar*(*x*, *r*)
{El puntero *r* señala la raíz de un árbol de búsqueda.
La función busca el valor *x* en este árbol
y devuelve un puntero al nodo que contiene a *x*.
Si *x* no está, la función devuelve **nulo**.}
si *r* = **nulo entonces** {*x* no está en el árbol}
devuelve nulo
sino si x = *r* ↑ *.valor* **entonces devolver** *r*
sino si *x* < *r* ↑ *.valor* **entonces devolver** *buscar*(*x*, *r* ↑ *.hijo-izquierdo*)
sino devolver *buscar*(*x*, *r.* ↑ *hijo-derecho*)

Aquí suponemos que el árbol de búsqueda está formado por nodos del tipo *nodo-binario* descrito anteriormente. A efectos de eficiencia, puede ser mejor reescribir el algoritmo evitando las llamadas recursivas, aunque algunos compiladores eliminan automáticamente este tipo de recursividad. El problema 5.11 le invita a hacer esto.

Resulta sencillo actualizar un árbol de búsqueda, esto es, borrar un nodo o añadir un nuevo valor, sin destruir la propiedad del árbol de búsqueda. Sin embargo, si se hace con descuido, el árbol resultante puede volverse *desequilibrado*. Con ello indicamos que muchos de los nodos pueden tener un solo hijo, así que sus ramas se vuelven largas y delgadas. Cuando sucede esto, ya no resulta eficiente hacer búsquedas en el árbol. En el caso peor, todos los nodos del árbol pueden tener exactamente un hijo, salvo una única hoja que no poseerá hijos. Con un árbol desequilibrado de estas características, buscar un valor en un árbol con n nodos puede implicar compararlo con el contenido de los n nodos.

Existe toda una gama de métodos para mantener equilibrado el árbol, y por tanto para garantizar que operaciones tales como las búsquedas o la adición o borrado de nodos requieran un tiempo que esté en $O(\log n)$ en el caso peor, donde n es el número de nodos que haya en el árbol. Estos métodos permiten también la implementación eficiente de varias operaciones adicionales. Entre las técnicas más antiguas se cuentan el uso de árboles AVL y de árboles 2-3; entre las sugerencias más modernas se cuentan los árboles rojinegros y los árboles biselados. Dado que estas estructuras no se utilizan en el resto del libro, nos limitaremos a mencionar su existencia.

Altura, profundidad y nivel. Resulta fácil confundir los términos que se utilizan para describir la posición de un nodo dentro de un árbol con raíz. La altura y el nivel por ejemplo son conceptos similares, pero no idénticos:

- La *altura* de un nodo es el número de aristas que hay en el camino más largo que vaya desde el nodo en cuestión hasta una hoja.
- La *profundidad* de un nodo es el número de aristas que haya en el camino que va desde el nodo raíz hasta el nodo en cuestión.
- El *nivel* de un nodo es igual a la altura de la raíz del árbol menos la profundidad del nodo estudiado.

Nodo	Altura	Profundidad	Nivel
alfa	2	0	2
beta	1	1	1
gamma	0	1	1
delta	0	2	0
épsilon	0	2	0
zeta	0	2	0

Figura 5.10. Altura, profundidad y nivel

Por ejemplo, la figura 5.10 da la altura, profundidad y nivel de todos los nodos del árbol ilustrado en la figura 5.5. Informalmente, si se dibuja el árbol con las sucesivas generaciones en capas bien ordenadas, entonces la profundidad de un nodo se halla numerando las capas hacia abajo, partiendo de 0 en la raíz; el nivel de un nodo se halla numerando las capas hacia arriba, y partiendo de 0 en la parte inferior; sólo la altura resulta algo más complicada.

Finalmente, definiremos la *altura del árbol* como la altura de su raíz; ésta es además la profundidad de la hoja más profunda y el nivel de la raíz.

5.6 TABLAS ASOCIATIVAS

Una *tabla asociativa* es exactamente igual que una matriz, salvo que su índice no está restringido a encontrarse entre dos cotas predeterminadas. Por ejemplo, si T es una tabla, se puede utilizar $T[1]$ y después $T[10^6]$ sin necesidad de reservar un millón de posiciones de almacenamiento para la tabla. Y aún hay algo mejor, se pueden utilizar cadenas en lugar de números como índice, así que T["Juan"] es tan legítimo como $T[1]$. Idealmente, una tabla no debería de ocupar mucho más espacio que el necesario para anotar los índices utilizados hasta el momento, junto al espacio para los valores almacenados en esas posiciones de la tabla.

La comodidad de las tablas tiene su precio: a diferencia de las matrices, las tablas no se pueden implementar de tal manera que se garantice que todas las búsquedas requieran un tiempo constante. La forma más sencilla de implementar una tabla es emplear una lista:

```
tipo lista_tabla = ↑nodo_tabla
tipo nodo_tabla = registro
        índice: tipo_índice
        valor: información
        siguiente: ↑nodo_tabla
```

Con esta implementación, el acceso a T["Juan"] se consigue recorriendo la lista hasta que se encuentre "Juan" en el campo *índice* de un nodo, o hasta que se alcance el final de la lista. En el primer caso, la información asociada se encuentra en el campo *valor* del mismo nodo; en el segundo, sabemos que "Juan" no se encuentra en la lista. Se van creando entradas nuevas a medida que se necesitan; véase el problema 5.12. Esta implementación es muy poco eficiente. En el caso peor, todas las solicitudes corresponden a elementos que faltan, lo cual nos obliga a explorar la lista completa en todos los accesos. Una secuencia de n accesos requiere, por tanto, un tiempo que se encuentra en $\Omega(n^2)$ en el caso peor. Si se utilizan hasta m índices diferentes en estos n accesos, con $m \leq n$, y si todas las solicitudes tienen igual probabilidad de acceder a cualquiera de estos índices, el rendimiento de esta implementación en el caso medio es tan malo como en el caso por: $\Omega(mn.)$ Siempre y cuando los campos *índice* se puedan comparar unos con otros y con el índice solicitado en un tiempo unitario, se pueden utilizar *árboles equilibrados* para reducir este tiempo a O (n log m) en el caso peor; véase la sección 5.5. Esto es

mejor, pero sigue sin ser suficientemente bueno para una de las aplicaciones principales de las tablas asociativas, a saber, los compiladores.

Prácticamente todos los compiladores utilizan una tabla asociativa para implementar la *tabla de símbolos*. Esta tabla contiene todos los identificadores que se utilizan en el programa que hay que compilar. Si *Juan* es un identificador, el compilador debe ser capaz de acceder a información relevante tal como su tipo, y el nivel en el que está definido. Empleando tablas, esta información se almacena sencillamente en T["Juan"]. El uso de cualquiera de las implementaciones esbozadas anteriormente ralentizaría al compilador de forma inadmisible cuando se enfrentase a programas que contuvieran un elevado número de identificadores. En lugar de hacer esto, la mayoría de los compiladores utiliza una técnica conocida con el nombre de *codificación dispersiva (hash coding)* o simplemente *dispersión (hashing)*. A pesar de que tiene un caso peor desastroso, la dispersión tiene en la práctica un rendimiento razonablemente bueno en casi todas las ocasiones.

Sea U el universo de índices potenciales que hay que implementar para la tabla asociativa, y sea $N << |U|$ un parámetro seleccionado según se describe a continuación. Una *función de dispersión* es una función h: $U \rightarrow \{0, 1, 2, \ldots, N-1\}$. Esta función debería dispersar eficientemente todos los índices probables: $h(x)$ debería ser diferente de $h(y)$ para la mayoría de los pares $x \neq y$ que tengan probabilidades de utilizarse simultáneamente. Por ejemplo, $h(x) = x \bmod N$ es razonablemente bueno a efectos de un compilador siempre y cuando N sea primo y x se obtenga a partir del nombre del identificador empleando cualquier representación entera estándar de las cadenas de caracteres, tal como ASCII.

Cuando $x \neq y$ pero $h(x) = h(y)$, decimos que se ha producido una *colisión* entre x e y. Si las colisiones son muy poco probables, podríamos implementar la tabla asociativa T con una matriz ordinaria $A[0..N\ 1]$, empleando $A[h(x)]$ cada vez que se necesitase $T[x]$. El problema es que $T[x]$ y $T[y]$ se confunden cada vez que colisionan x e y. Desafortunadamente, esto es intolerable, porque la probabilidad de colisión no se puede despreciar a no ser que $N >> m^2$, en donde m es el número de índices distintos que se están empleando realmente; véase el problema 5.14. Para esta dificultad se han propuesto muchas soluciones. La más sencilla es la *tabla de dispersión abierta* o *con encadenamiento*. Cada entrada de la matriz $A[0..N\ 1]$ es un elemento del tipo *lista*: $A[i]$ contiene la lista de todos los índices que al dispersarse llegan al valor i, junto con la información relevante. La figura 5.11 ilustra la situación después de hacer las cuatro solicitudes siguientes en la tabla asociativa T:

T["Laurel"] $\leftarrow 3$
T["Chaplin"] $\leftarrow 1$
T["Hardy"] $\leftarrow 4$
T["Keaton"] $\leftarrow 1$

en este ejemplo, $N = 6$, h("Keaton") = 1, h("Laurel") = h("Hardy") = 2 y h("Chaplin") = 4.

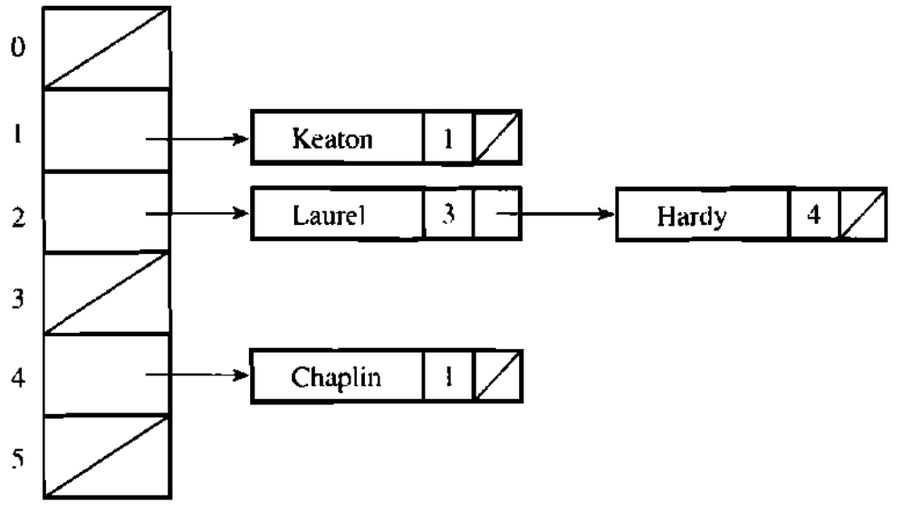


Figura 5.11. Ilustración de una dispersión (hashing)

El *factor de carga* de la tabla es m/N, en donde m es el número de índices distintos que se han almacenado en la tabla y N es el tamaño de la matriz que se emplea para implementarla. Si suponemos que todos los índices, todos los valores almacenados en la tabla y todos los punteros ocupan una cantidad de espacio constante, entonces la tabla ocupa un espacio que está en $\Theta(N + m)$ y la longitud media de la lista es igual al factor de carga. Por tanto, un incremento de N reduce la longitud media de la lista pero incrementa el espacio ocupado por la tabla. Si se mantiene el factor de carga entre un medio y 1, la tabla ocupa un espacio que está en $\Theta(m)$, lo cual es óptimo salvo un pequeño factor constante, y la longitud media de la lista es menor que uno, lo cual tiene probabilidades de implicar un acceso eficiente a la tabla. Resulta tentador mejorar este esquema sustituyendo las N listas de colisión por árboles equilibrados, pero esto no merece la pena si el factor de carga se mantiene en valores pequeños, a no ser que resulte esencial mejorar el rendimiento en el caso peor.

El factor de carga se puede mantener en valores pequeños mediante una *redispersión*. Teniendo en mente la aplicación del compilador, por ejemplo, el valor inicial de N se toma de tal modo que esperamos que los programas pequeños utilicen menos de N identificadores distintos. Se permite que el factor de carga sea menor que un medio cuando el número de identificadores sea pequeño. Cuando se encuentran más de N identificadores diferentes, lo cual da lugar a que el factor de carga supere el valor uno, ha llegado el momento de doblar el tamaño de la matriz empleada para implementar la tabla de dispersión. En ese momento, la función de dispersión debe cambiar para doblar su alcance, y todas las entradas que estén ya en la tabla deben ser redistribuidas a sus nuevas posiciones en alguna lista de la matriz mayor. La redistribución es costosa, pero es tan infrecuente que no da lugar a un incremento dramático del tiempo *amortizado* que se necesita por cada acceso a la tabla. Se repite la redistribución cada vez que el factor de carga sobrepasa el valor uno; después de la reorganización, el factor de carga vuelve a caer hasta el valor un medio.

Desafortunadamente, una lista con una longitud media baja no garantiza un tiempo medio de acceso reducido. El problema es que cuanto más larga sea la

lista, más probable es que se acceda a uno de sus elementos. Por tanto, los casos malos tienen más probabilidades de aparecer que los casos buenos. En un caso extremo, podría haber una lista de longitud N y $N-1$ listas de longitud cero. Aun cuando la longitud media de la lista sea 1, la situación no es mejor que si empleásemos el enfoque sencillo de una *lista_tabla*. Si la tabla se está utilizando en un compilador, sucedería si todos los identificadores de un programa se distribuyeran a un mismo índice. Aunque esto es improbable, no podemos descartarlo. Sin embargo, se puede demostrar que todo acceso requiere un tiempo esperado constante en el sentido amortizado, siempre y cuando se efectúe una redistribución cada vez que el factor de carga sobrepase el valor 1, y siempre y cuando hagamos la suposición (poco natural) consistente en que todos los identificadores posibles tienen igual probabilidad de ser utilizados. En la práctica, la dispersión funciona bien en casi todas las ocasiones, aun cuando los identificadores no se seleccionen al azar. Además, veremos en la Sección 10.7.3 la forma de eliminar esta suposición acerca de la distribución de probabilidades de los casos que hay que resolver, manteniendo un rendimiento esperado demostrablemente bueno.

5.7 MONTÍCULOS (HEAPS)

Un montículo *(heap)* es un tipo especial de árbol con raíz que se puede implementar eficientemente en una matriz sin ningún puntero explícito. Esta estructura tan interesante se presta a numerosas aplicaciones, incluyendo una notable técnica de ordenación denominada *ordenación por el método del montículo (heapsort)*, que se presentará más adelante en esta sección. También se puede utilizar para representar eficientemente ciertas listas dinámicas de prioridad, tales como la lista de sucesos en una simulación de una lista de tareas que haya de ser planificada por un sistema operativo.

Se dice que un árbol binario es *esencialmente completo* si todo nodo interno, con la posible excepción de un nodo especial, tiene exactamente dos hijos. El nodo especial, si existe uno, está situado en el nivel 1, y posee un hijo izquierdo pero no tiene hijo derecho. Además, o bien todas las hojas se encuentran en el nivel 0, ó bien están en los niveles cero y uno, y ninguna hoja del nivel 1 está a la izquierda de un nodo interno del mismo nivel. Intuitivamente, un árbol esencialmente completo es uno en el que los nodos internos se han subido en el árbol lo más posible, con los nodos internos del último nivel empujados hacia la izquierda; las hojas llenan el último nivel que contiene nodos internos, si es que queda algún espacio, y después se desbordan hacia la izquierda en el nivel cero. Por ejemplo, la figura 5.12 ilustra un árbol binario esencialmente completo que contiene 10 nodos. Los cinco nodos internos ocupan el nivel 3 (la raíz), el nivel 2 y el lado izquierdo del nivel 1; las cinco hojas llenan el lado derecho del nivel uno y continúan después por la izquierda del nivel cero.

Si un árbol binario esencialmente completo tiene una altura k, entonces hay un nodo (la raíz) en el nivel k, dos nodos en el nivel $k-1$, y así sucesivamente; hay $2^k - 1$

nodos en el nivel 1, y al menos uno pero no más de 2^k en el nivel 0. Si el árbol contiene n nodos en total, contando tanto los nodos internos como las hojas, se sigue que $2^k \leq n \leq 2^{k+1}$. Análogamente, la altura de un árbol que contiene n nodos es $k = \lfloor \lg n \rfloor$, resultado éste que utilizaremos posteriormente.

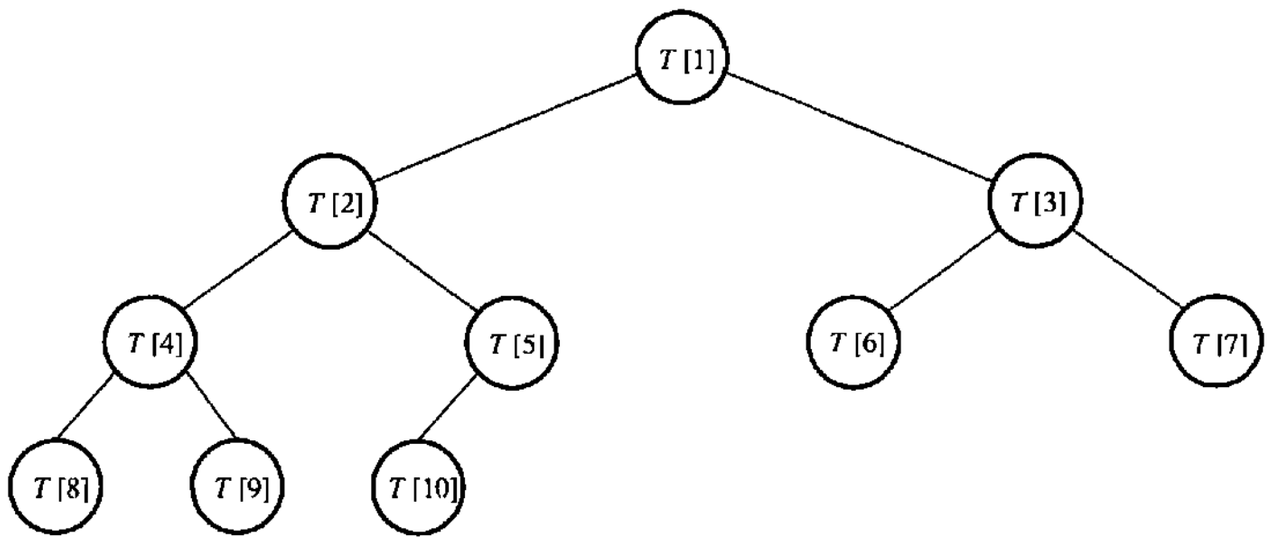


Figura 5.12. Un árbol binario esencialmente completo

Esta clase de árbol se puede representar en una matriz T poniendo los nodos de profundidad k, de izquierda a derecha, en las posiciones $T[2^k]$, $T[2^k+1],\ldots,T[2^{k+1}-1]$, con la posible excepción del nivel cero, que puede estar incompleto. La figura 5.12 indica los elementos de la matriz que corresponden a cada nodo del árbol. Utilizando esta representación, el padre del nodo representado en $T[i]$ se encuentra en $T[i \div 2]$ para $i > 1$ (la raíz $T[1]$ carece de padre), y los hijos del nodo representado en $T[i]$ se encuentran en $T[2i]$ y $T[2i+1]$ siempre que existan. El sub-árbol cuya raíz está en $T[i]$ también es fácil de identificar.

Un *montículo* es un árbol binario esencialmente completo, cada uno de cuyos nodos incluye un elemento de información denominado *valor del nodo*, y que tiene la propiedad consistente en que el valor de cada nodo interno es mayor o igual que los valores de sus hijos. Esto se llama *propiedad del montículo*. La figura 5.13 muestra un ejemplo de montículo con 10 nodos. El árbol subyacente es, por supuesto, el que se muestra en la figura 5.12, pero ahora hemos marcado cada nodo con su valor. La propiedad del montículo se puede comprobar fácilmente. Por ejemplo, el nodo cuyo valor es 9 tiene dos hijos cuyos valores son 5 y 2: los dos hijos tienen un valor menor que el valor de su padre. Este mismo montículo se puede representar mediante la matriz siguiente:

10	7	9	4	7	5	2	2	1	6

Dado que el valor de cada nodo interno es mayor o igual que los valores de sus hijos, que a su vez tienen valores mayores o iguales que los valores de *sus* hijos, y

así sucesivamente, la propiedad del montículo asegura que el valor de todo nodo interno es mayor o igual que los valores de todos los nodos que yacen en los subárboles situados por debajo de él. En particular, el valor de la raíz es mayor o igual que los valores de todos los demás nodos del montículo.

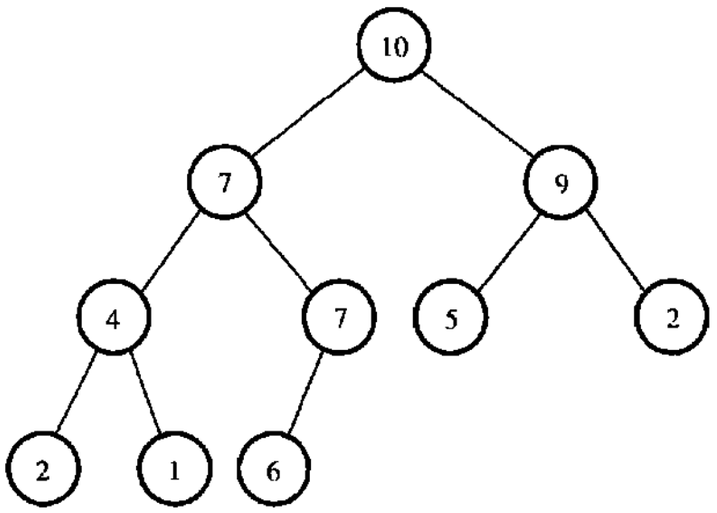


Figura 5.13 Un montículo (heap)

La característica crucial de esta estructura de datos es que la propiedad del montículo se puede restaurar de forma eficiente si se modifica el valor de un nodo. Si el valor de un nodo crece hasta llegar a ser mayor que el valor de su padre, basta con intercambiar estos dos valores, y continuar entonces el mismo proceso hacia arriba si es necesario hasta que se haya restaurado la propiedad del montículo. Diremos que el valor modificado ha *flotado* hasta su nueva posición dentro del montículo. (Esta operación suele denominarse *flotar*.) Por ejemplo, si se modifica el valor 1 de la figura 5.13 para pasar a ser 8, entonces podemos restaurar la propiedad del montículo intercambiando el 8 con su padre, que es 4, y volviendo

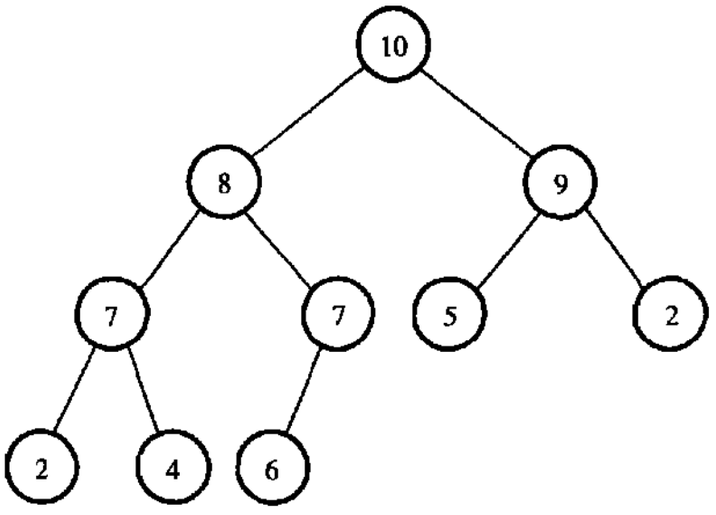


Figura 5.14 El montículo, después de flotar el 8 hasta su lugar

a intercambiarlo con su nuevo padre, que es 7, obteniendo así el resultado que se muestra en la figura 5.14.

Si por el contrario el valor del nodo decrece de tal modo que pasa a ser menor que el valor de alguno de sus hijos, entonces basta intercambiar el valor modificado con el mayor de los valores de los hijos, y continuar este proceso hacia abajo si es necesario hasta que se restaure la propiedad del montículo.

Diremos que el valor modificado se ha *hundido* hasta su nueva posición. Por ejemplo, si el 10 de la raíz de la figura 5.14 pasa a ser un 3, podemos restaurar la propiedad del montículo intercambiando el 3 con su hijo mayor, el 9, e intercambiándolo una vez más con el mayor de sus hijos nuevos, el 5. El resultado que se obtiene aparece en la figura 5.15.

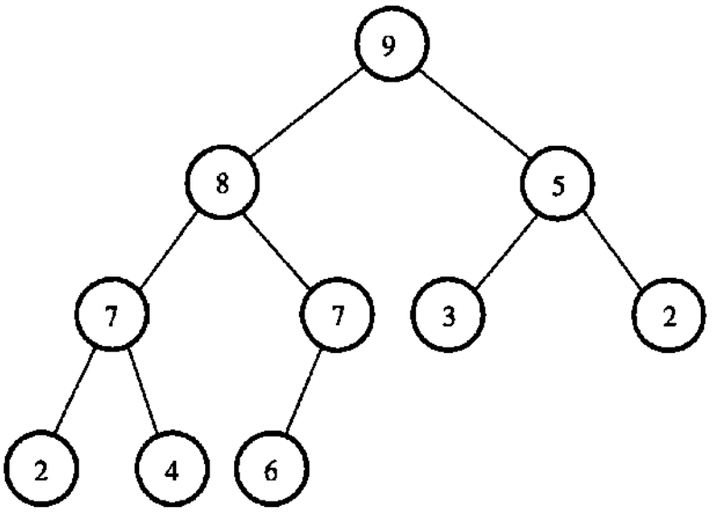


Figura 5.15. El montículo, después de hundir el 3 hasta su lugar

Los procedimientos siguientes describen de manera más formal los procesos básicos para manipular un montículo. Por claridad, se han escrito de tal modo que reflejen en lo posible la discusión anterior. Si tiene usted intención de utilizar montículos para una aplicación «real», le animamos a que determine la forma de evitar la ineficiencia causada por nuestra utilización de la instrucción «intercambiar».

```
procedimiento modificar-montículo(T[1..n], i, v)
  {T[1..n] es un montículo. T[i] recibe el valor v, y se vuelve a
  establecer la propiedad del montículo. Suponemos que 1 ≤ i ≤ n.}
  x ← T[i]
  T[i] ← v
  si v < x entonces hundir(T, i)
               sino flotar(T, i)
```

```
procedimiento hundir(T[1..n], i)
    {Este procedimiento hunde el nodo i para restablecer
       la propiedad del montículo en T[1..n]. Suponemos que T sería
    un montículo si T[i] fuera suficientemente grande. También
    suponemos que 1 ≤ i ≤ n.}
    k ← i
    repetir
        j ← k
        {buscar el hijo mayor del nodo j}
        si 2j ≤ n y T[2j] > T[k] entonces k ← 2j
        si 2j < n y T[2j + 1] > T[k] entonces k ← 2j + 1
        intercambiar T[j] y T[k]
        {si j = k, entonces el nodo ha llegado a su posición final}
    hasta que j = k
procedimiento flotar(T[1..n], i)
    {Este procedimiento filtra el nodo i para restablecer la
       propiedad del montículo en T[1..n]. Suponemos que T sería
       un montículo si T[i] fuera suficientemente pequeño. También
       suponemos que 1 ≤ i ≤ n
       El parámetro n no se utilizará aquí}
       k ← i
       repetir
           j ← k
           si j > 1 y T[j ÷ 2] < T[k] entonces k ← j ÷ 2
           intercambiar T[j] y T[k]
           {si j = k, entonces el nodo ha llegado a su posición final}
       hasta que j = k
```

El montículo es una estructura ideal para hallar el mayor de los elementos de un conjunto, para eliminarlo, para añadir un nodo nuevo o para modificar un nodo. Éstas son exactamente las operaciones que se necesitan para implementar eficientemente las listas de prioridad dinámica: el valor del nodo da la prioridad del suceso correspondiente, el suceso de prioridad más alta se encuentra siempre en la raíz del montículo, y la prioridad de un suceso se puede modificar dinámicamente en cualquier momento. Esto resulta especialmente útil en las simulaciones por computadora y en el diseño de planificadores para sistemas operativos. Se ilustran más abajo algunos procedimientos típicos.

```
función buscar-max(T[1..n])
    {Proporciona el mayor elemento del montículo T[1..n]}
    devolver T[1]
procedimiento borrar-max(T[1..n])
    {Elimina el mayor elemento del montículo T[1..n]
       y restaura la propiedad del montículo en T[1..n-1]}
    T[1] ← T[n]
    hundir(T[1..n-1], 1)
```

procedimiento *añadir-nodo*($T[1..n]$, v)
  {Añade un elemento cuyo valor es v al nodo $T[1..n]$
    y restaura la propiedad del montículo en $T[1..n\text{-}1]$}
  $T[n+1] \leftarrow v$
  flotar($T[1..n+1]$, $n+1$)

Queda por ver la forma de crear un montículo partiendo de una matriz $T[1..n]$ de elementos que estén en un orden indefinido. La solución evidente consiste en partir de un montículo vacío y añadir elementos uno por uno.

procedimiento *crear-montículo-lento*($T[1..n]$)
  {Este procedimiento transforma la matriz $T[1..n]$
    en un montículo, aunque de forma más bien ineficiente}
  para $i \leftarrow 2$ **hasta** n **hacer** *flotar* ($T[1..i]$, i)

Sin embargo, esta aproximación no es particularmente eficiente; véase el problema 5.19. Existe un algoritmo más inteligente para hacer un montículo. Supongamos, por ejemplo, que nuestro punto de partida es la matriz siguiente:

1	6	9	2	7	5	2	7	4	10

que representa al árbol de la figura 5.16a. Primero convertimos en montículos aquellos sub-árboles cuyas raíces se encuentren en el nivel 1; esto se hace sumergiendo estas raíces, según se ilustra en la figura 5.16b. Los sub-árboles de nivel inmediatamente superior se transforman entonces en montículos, sumergiendo una vez más sus raíces. La figura 5.16c muestra el proceso para el sub-árbol izquierdo. Esto da lugar a un árbol binario esencialmente completo que corresponde a la matriz

1	10	9	7	7	5	2	2	4	6

Sólo queda sumergir su raíz para obtener el montículo deseado. El proceso final, entonces, queda en la forma siguiente:

10	1	9	7	7	5	2	2	4	6
10	7	9	1	7	5	2	2	4	6
10	7	9	4	7	5	2	2	1	6

La representación en árbol de la forma final de la matriz es la que se mostró previamente como figura 5.13.

He aquí una descripción formal del algoritmo.

procedimiento *crear-montículo*($T[1..n]$)
{Este procedimiento transforma la matriz
$T[1..n]$ en un montículo}
para $i \leftarrow \lfloor n/2 \rfloor$ **bajando hasta** 1 **hacer** *hundir*(T, i)

Teorema 5.7.1 *El algoritmo construye un montículo en tiempo lineal*

Demostración. Daremos dos demostraciones de esta afirmación:

1. Como barómetro, utilizamos las instrucciones del bucle **repetir** del algoritmo *hundir*. Para hundir un nodo hasta el nivel r, está claro que se hacen como mucho $r + 1$ pasadas por el bucle. Ahora bien, la altura de un montículo que contiene n nodos es, digamos, $\lfloor \lg n \rfloor = k$. En el montículo hay 2^{k-1} nodos en el nivel 1 (de los cuales no todos son necesariamente nodos internos), 2^{k-2} nodos en el nivel 2 (todos los cuales deben ser nodos internos) y así sucesivamente, hasta un nodo en el nivel k, que es la raíz. Cuando se aplica *crear-montículo* hundimos el valor de todos los nodos internos. Por tanto, si t es el número total de pasadas por el bucle repetir, tenemos

$$\begin{aligned} t &\leq 2\times 2^{k-1} + 3\times 2^{k-2} + \ldots + (k+1)\times 2^{0} \\ &< -2^{k} + 2^{k-1}(2^{-1} + 2\times 2^{-2} + 3\times 2^{-3} + \ldots) \\ &= 2^{k+2} - 2^{k} < 4n \end{aligned}$$

donde se ha utilizado la Proposición 1.7.12 para sumar la serie infinita. El trabajo requerido está, por tanto, en $O(n)$.

2. Sea $t(k)$ el tiempo requerido en el caso peor para construir un montículo de altura k como máximo. Supongamos que $k \geq 2$. Para construir el montículo, el algoritmo transforma primero los dos sub-árboles asociados a la raíz en árboles de altura k–1 como máximo. (El sub-árbol derecho podría ser de altura k–2.) Entonces el algoritmo hunde la raíz por una ruta cuya longitud es k como máximo, lo cual requiere un tiempo $s(k) \in O(k)$ en el caso peor. Así se obtiene la recurrencia asintótica

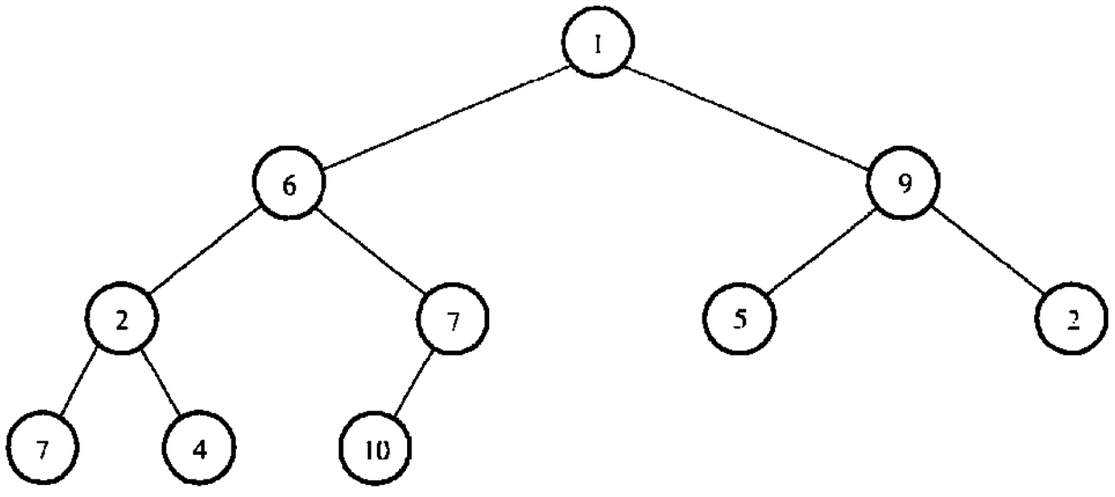
$$t(k) \leq 2t(k-1) + s(k)$$

Esta recurrencia es parecida al ejemplo 4.7.7, que proporciona $t(k) \in O(2^k)$. Pero un montículo que contenga n elementos tiene como altura $\lfloor \lg n \rfloor$, así que se puede construir en t ($\lfloor \lg n \rfloor$) pasos, lo cual está en $O(n)$ porque $2^{\lfloor \lg n \rfloor} \leq n$.

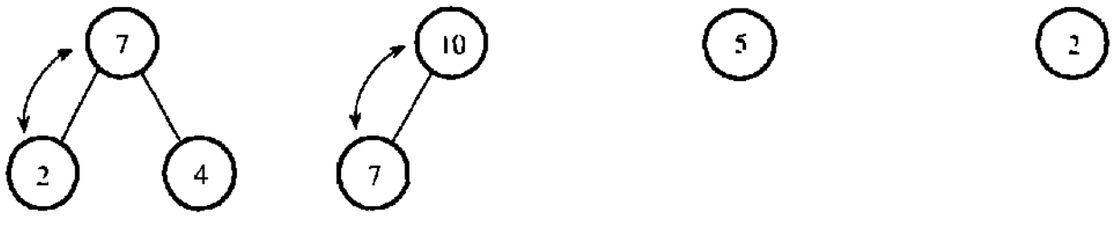
Williams inventó el montículo para que sirviera como estructura de datos subyacente al siguiente algoritmo de ordenación:

procedimiento *ordenación por montículo*($T[1..n]$)
  {T es la matriz que hay que ordenar}
  crear-montículo(T)
  para $i \leftarrow n$ **bajando hasta** 2 **hacer**
    intercambiar $T[1]$ y $T[i]$
    hundir($T[1..i-1]$, 1)

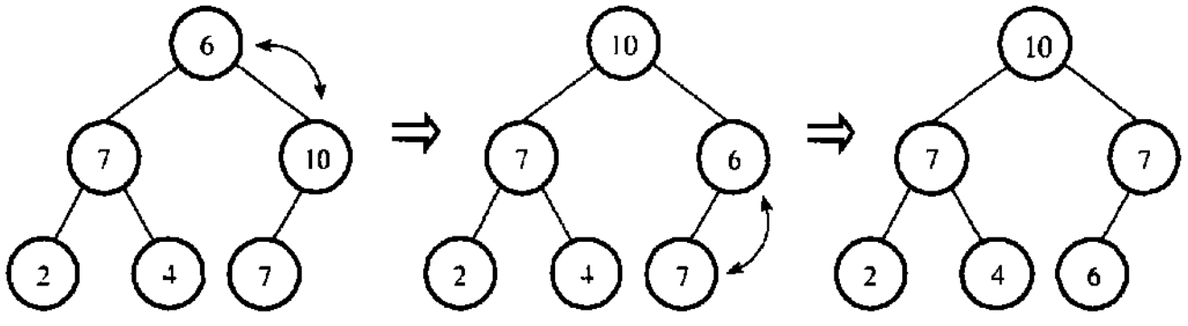
Teorema. 5.7.2 *El algoritmo requiere un tiempo que está en* $O(n \log n)$ *para ordenar n elementos*



a) La situación inicial



(b) Los sub-árboles de nivel 1 se convierten en montículos



(c) Se convierte un sub-árbol de nivel 2 en un montículo
(el otro ya es un montículo)

Figura 5.16. Creación de un montículo

Demostración. Sea $t(n)$ el tiempo necesario para ordenar una matriz de n elementos en el caso peor. La operación *crear-montículo* requiere un tiempo lineal en n, y la altura del montículo que se produce es $\lfloor \lg n \rfloor$. El bucle **para** se ejecuta $n-1$ veces. En cada pasada por el bucle, la instrucción «intercambiar» requiere un tiempo constante, y la operación *hundir* hunde la raíz a lo largo de un camino que tiene como máximo una longitud lg n, lo cual requiere un tiempo que es del orden de lg n en el caso peor. Consiguientemente:

$$t(n) \in O(n) + (n-1)O(1) + (n-1)O(\log n) = O(n \log n)$$

Se puede demostrar que $t(n) \in \Theta(n \log n)$: el orden *exacto* de $t(n)$ es también n log n, tanto en el caso peor como en el caso medio, suponiendo que todas las permutaciones iniciales de los objetos que hay que ordenar sean igualmente probables. Sin embargo, esto es más difícil de demostrar.

El concepto básico de un montículo se puede mejorar de diferentes maneras. Para aquellas aplicaciones que necesiten *flotar* con más frecuencia que *hundir* (véase, por ejemplo, el problema 6.16), es ventajoso tener más de dos hijos por nodo interno. Esto acelera *flotar* (porque el montículo es menos profundo) a costa de hacer más lentas las operaciones tales como *hundir*, que tienen que considerar todos los hijos de cada nivel. Sigue siendo posible representar esos montículos en una matriz sin punteros explícitos, pero hay que tener un poquito de cuidado para hacerlo bien; véase el problema 5.23.

Para las aplicaciones que tienden a hundir un nodo raíz actualizado hasta casi el nivel inferior, es ventajoso ignorar temporalmente el nuevo valor almacenado en la raíz, optando más bien por tratar a este nodo como si estuviese vacío, esto es, como si contuviese un valor más pequeño que cualquier otro valor del árbol. El nodo vacío, por tanto, se sumergirá hasta llegar finalmente a una hoja. En este momento, se vuelve a poner el valor relevante en la hoja vacía, y flota hasta su posición correcta. La ventaja de este procedimiento es que solamente necesita una comparación en cada nivel mientras se está sumergiendo el nodo vacío, en lugar de ser necesarias dos, como en el procedimiento habitual. (Esto se debe a que tenemos que comparar los hijos entre sí, pero no es necesario comparar el hijo mayor con su padre.) La experiencia muestra que este enfoque produce una mejora con respecto al algoritmo *ordenar por montículo* clásico.

En algunas ocasiones, será preciso utilizar un *montículo invertido*. Con esto denotamos un árbol binario esencialmente completo en el cual el valor de todo nodo interno es menor o igual que los valores de sus hijos, y no mayor o igual como en el montículo ordinario. En un montículo invertido, el ítem más pequeño se encuentra en la raíz. Todas las propiedades de un montículo ordinario son aplicables *mutatis mutandis*.

Aun cuando los montículos pueden utilizarse para implementar de forma eficiente la mayoría de las operaciones necesarias para manejar listas dinámicas de prioridad, hay algunas operaciones para las que no resultan ade-

cuados. Por ejemplo, no hay una buena manera de buscar un cierto elemento dentro de un montículo. En procedimientos tales como *hundir* y *flotar*, vistos más arriba, proporcionábamos la dirección del nodo implicado como uno de los parámetros del procedimiento. Además, no hay una forma eficiente de fusionar dos montículos de la clase que se ha descrito. Lo mejor que podemos hacer es poner los contenidos de ambos montículos uno junto a otro dentro de la matriz, y llamar entonces al procedimiento *crear-montículo* aplicándolo a la matriz.

Como se verá en la sección siguiente, no es difícil producir *montículos fusionables* a costa de una cierta complicación en las estructuras de datos utilizadas. Con todo, la búsqueda de un cierto elemento es ineficiente en cualquier tipo de montículo que se considere.

5.8 MONTÍCULOS BINOMIALES

En un montículo ordinario que contenga n elementos, buscar el mayor de ellos requiere un tiempo que está en $O(1)$. Borrar el mayor elemento, o insertar un nuevo elemento, requiere un tiempo que está en $O(\log n)$. Sin embargo, fusionar dos montículos que contengan entre los dos n elementos requiere un tiempo que está en $O(n)$. En esta sección describiremos un tipo distinto de montículo, en el cual la búsqueda del mayor elemento sigue requiriendo un tiempo en $O(1)$, y el borrado del mayor elemento sigue requiriendo un tiempo en $O(\log n)$. Sin embargo, la fusión de dos de estos nuevos montículos sólo requiere un tiempo en $O(\log n)$, y la inserción de un nuevo elemento —siempre y cuando se considere el tiempo amortizado, y no el coste en sí de cada operación— solamente requiere un tiempo en $O(1)$.

Definiremos primero los *árboles binomiales*. El i-ésimo árbol binomial B_i, con $i \geq 0$, se define recursivamente como aquel que consta de un nodo raíz con i hijos, en donde el j-ésimo hijo, $1 \leq j \leq i$, es a su vez la raíz de un árbol binomial B_{j-1}. La figura 5.17 muestra desde B_0 hasta B_4. Es fácil mostrar por inducción matemática que el árbol binomial B_i contiene 2^i nodos, de los cuales $\binom{i}{k}$ están a una profundidad k, $0 \leq k \leq i$. Aquí $\binom{i}{k}$ es el coeficiente binomial definido en la Sección 1.7.3. Ésta es, por supuesto, la razón del nombre a los *árboles binomiales*. Suponemos que todo nodo de estos árboles puede almacenar un valor. Dado que los nodos poseen un número variable de hijos, lo más probable es que la mejor manera de representarlos en una computadora sea utilizar el tipo *nodoarbol1* definido en la Sección 5.5.

Para definir un *montículo binomial*, comenzamos con una colección de árboles binomiales. Cada árbol binomial de la colección debe tener un tamaño diferente, y ademas todos ellos tienen que tener la propiedad del montículo: el valor almacenado en todo nodo interno debe ser mayor o igual que los valores de sus hijos. Esto asegura que el mayor elemento del montículo se encuentre en la raíz de uno de sus árboles binomiales. Para completar la defini-

ción de un montículo binomial, añadiremos punteros que unan cada raíz con la siguiente, por orden de tamaño creciente de los árboles binomiales. Es conveniente organizar las raíces en una lista doblemente enlazada para que la inserción o borrado de una raíz resulte sencilla. Por último, añadimos un puntero a la raíz que contenga el mayor elemento del montículo. La figura 5.18 ilustra un montículo binomial con 11 elementos. Es fácil ver que un montículo binomial que contenga n elementos consta como máximo de $\lceil \lg n \rceil$ árboles binomiales.

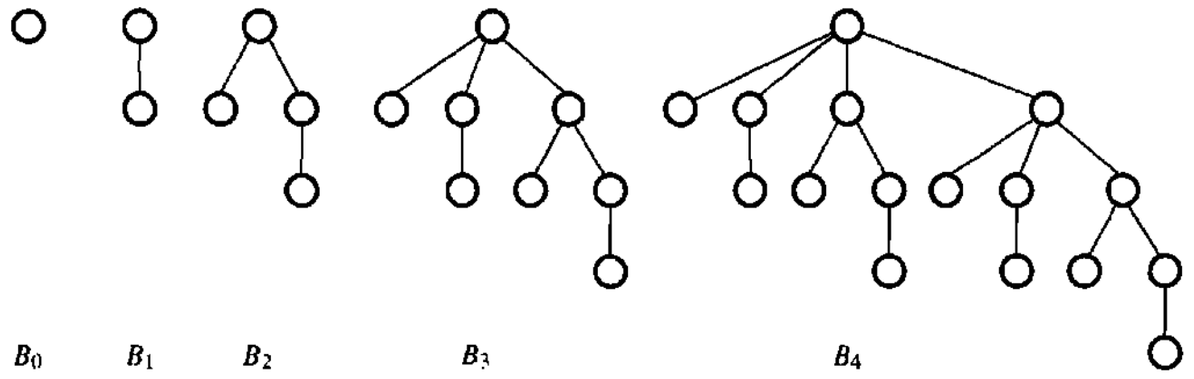


Figura 5.17. Árboles binomiales del B_0 al B_4.

Supongamos que se tienen dos árboles binomiales B_i, del mismo tamaño pero conteniendo posiblemente valores distintos. Supóngase que ambos tienen la propiedad del montículo. Entonces es fácil combinarlos en un único árbol binomial B_{i+1}, que seguirá poseyendo la propiedad del montículo. Se hace que la raíz de menor valor sea el $(i + 1)$-ésimo hijo de la raíz que tenga el mayor valor, y hemos terminado. La figura 5.19 muestra la forma en que dos B_2 se pueden combinar para formar un B_3 de esta manera. Llamaremos a esta operación *enlazado* de dos árboles binomiales. Está claro que requiere un tiempo en $O(1)$.

A continuación se describe la forma de fusionar dos montículos binomiales H_1 y H_2. Cada uno de ellos consta de una colección de árboles binomiales dispuestos por orden creciente de tamaños. Comenzamos por determinar los posibles B_0 que puedan estar presentes. Si ni H_1 ni H_2 contienen un B_0, entonces no hay nada que hacer en esta fase. Si sólo H_1 o H_2 incluyen un B_0, lo reservamos para que forme parte del resultado. En caso contrario, se enlazan los dos B_0 para formar un B_1. A continuación, se determina si está presente algún B_1. Podemos tener que enfrentarnos a un máximo de tres; de H_1 y H_2, uno cada uno, y otro que «nos llevamos» de la fase anterior. Si no hay ninguno entonces no hay nada que hacer en esta fase. Si sólo hay uno, lo reservamos para que forme parte del resultado. En caso contrario, se enlazan dos cualesquiera de los B_1 para formar un B_2; si existe un tercero, se reserva para que forme parte del resultado. A continuación se buscan los B_2 que pudieran estar presentes, y así sucesivamente.

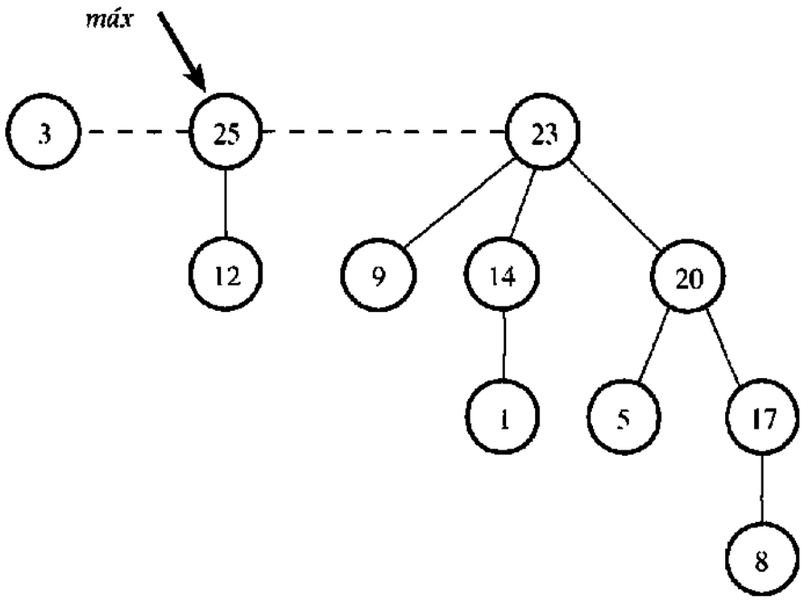


Figura 5.18. Un montículo binomial que contiene 11 elementos

En general, al llegar a la fase *i* se tienen hasta tres B_i entre manos, uno de cada uno de H_1 y H_2, y otro que nos «llevamos» de la fase anterior. Si no hay ninguno, entonces no hay nada que hacer en esa fase; si hay sólo uno, lo reservamos para formar parte del resultado; en caso contrario, enlazamos dos cualesquiera de los B_i para formar un B_{i+1}, y reservamos el tercero, si existiera, para formar parte del resultado. Si se forma un B_{i+1} en esta fase, entonces se «lleva uno» a la fase siguiente. A medida que vayamos avanzando, uniremos las raíces de aquellos árboles binomiales que haya que mantener, y además llevaremos la cuenta de la raíz que tenga el mayor de los valores para poder dar valor al puntero correspondiente en el resultado. La figura 5.20 muestra la forma en que se pueden fusionar un montículo binomial de 6 elementos y otro de 7 para formar un montículo binomial de 13 elementos.

La analogía con la suma binaria es bastante próxima. En ésta, se tienen en cada etapa hasta tres unos, uno que se «lleva» de la posición anterior, y uno de cada uno de los operandos. Si no hay ninguno, el resultado contiene un 0 en esta posición; si hay un solo 1 el resultado contiene un 1; en caso contrario, dos de los unos generan un 1 que se lleva a la posición siguiente, y el resultado de esa posición es un 1 ó un 0, dependiendo de si estaban presentes 2 ó 3 unos inicialmente.

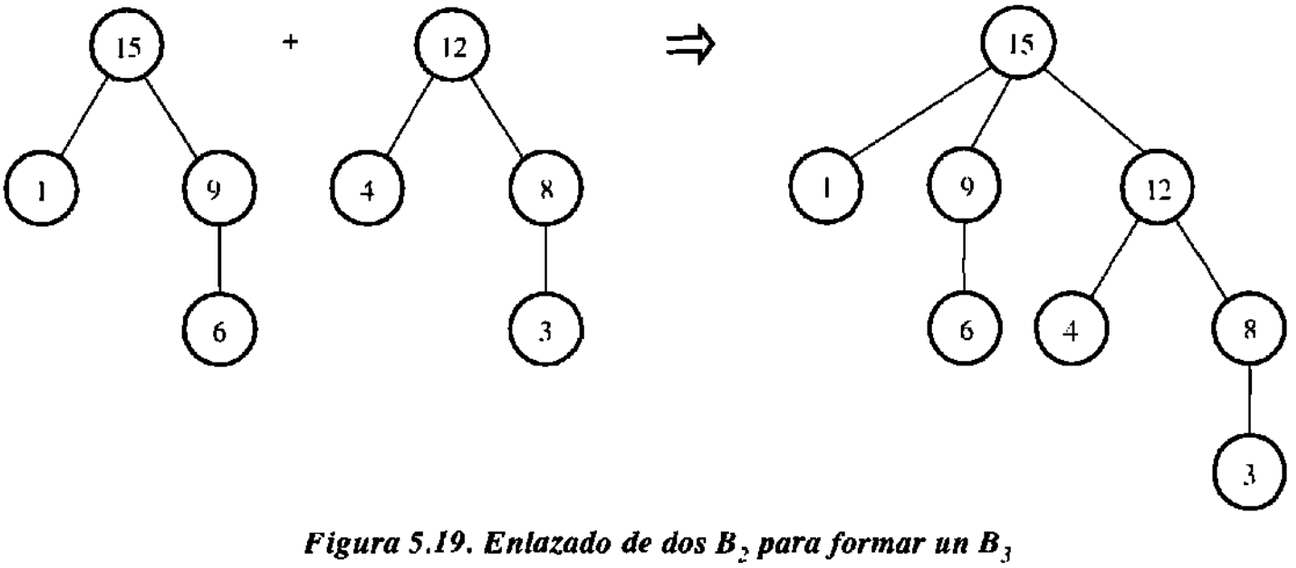


Figura 5.19. Enlazado de dos B_2 para formar un B_3

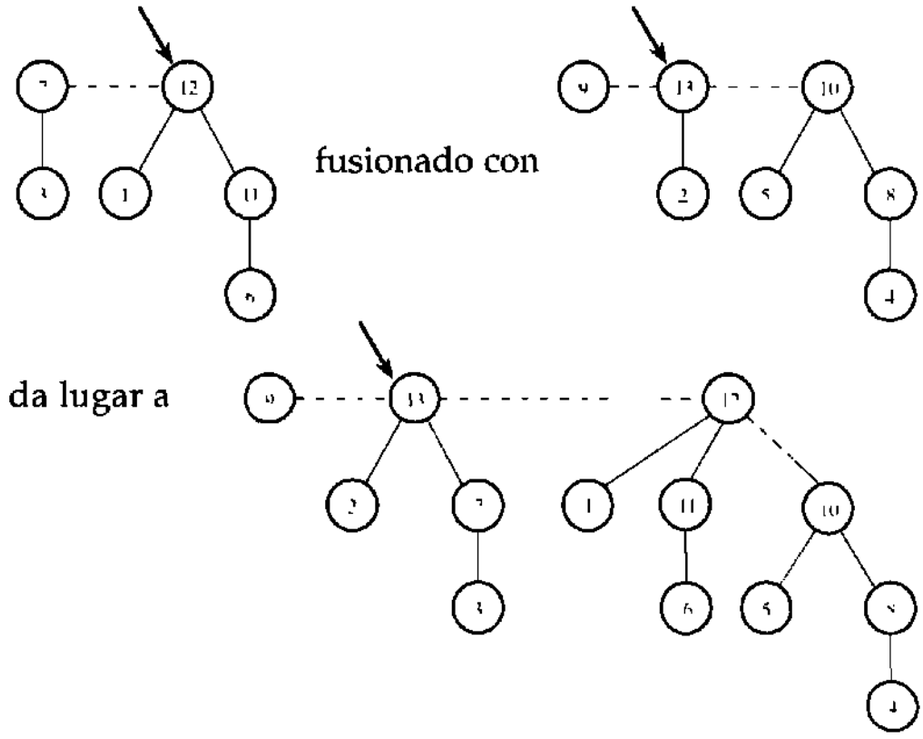


Figura 5.20. Fusión de dos montículos binomiales

Si el resultado de una operación de fusión es un montículo binomial que consta de n elementos, es posible construirlo en un máximo de $\lfloor \lg n \rfloor + 1$ etapas. Cada etapa requiere como mucho una operación de enlazado, más el ajuste de unos pocos punteros. Por tanto, se puede efectuar una etapa en un tiempo $O(1)$, y la fusión completa requiere un tiempo que está en $O(\log n)$.

La búsqueda del máximo elemento de un montículo binomial consiste simplemente en proporcionar el elemento indicado por el puntero correspondiente. Claramente, esto se puede hacer en un tiempo que está en $O(1)$. El borrado del máximo elemento de un montículo binomial H se hace en la forma siguiente:

(i) Sea B el árbol binomial cuya raíz contiene el mayor elemento de H. Se elimina B de H, enlazando los elementos constituyentes restantes de H en un nuevo montículo binomial H_1.
(ii) Se elimina la raíz de B. Se fusionan los árboles binomiales que antes fueran los sub–árboles de esta raíz en un nuevo montículo binomial B_2. (Ya está en orden de tamaños crecientes, según lo requerido).
(iii) Se fusionan H_1 y H_2 en un único montículo binomial. Este es el resultado solicitado.

Aquí el paso (i) requiere un tiempo que está en $O(1)$, puesto que no requiere más que el ajuste de unos pocos punteros. El paso (ii) requiere un paso que está en $O(\log n)$, porque la raíz de cualquier árbol de H tiene como máximo $\lfloor \log n \rfloor$ hijos; este tiempo exacto depende de la forma en que estén enlazados entre sí los hijos. El paso (iii) requiere un tiempo que está en $O(\log n)$, como acabamos de ver. Por tanto, la operación completa requiere un tiempo que está en $O(\log n)$.

La inserción de un elemento nuevo en un montículo binomial H se puede hacer en la forma siguiente:

(i) Se toma el elemento que haya que insertar y se transforma en un árbol binomial que contenga únicamente ese elemento. Lo único que se necesita es crear un único nodo, e inicializar correctamente su valor. Llamaremos B_0^* a este nuevo árbol binomial.

(ii) Se hace $i \leftarrow 0$.

(iii) Si H contiene a un B_i, entonces:
se elimina la raíz de este B_i de la lista de raíces de H;
se enlaza B_i^* con el B_i de H, para formar un árbol binomial B_{i+1}^*;
se hace $i \leftarrow i + 1$;
se repite el paso (iii);
En caso contrario, se va al paso (iv).

(iv) Se inserta la raíz de B_i^* en la lista de raíces que pertenecen a H. Si el elemento que se acaba de insertar es mayor que ningún otro elemento de H, se hace que el puntero de H señale la raíz de B_i^*.

La operación completa nos recuerda el incremento de un contador binario: en la mayor parte de las ocasiones, el paso (iii) se ejecuta sólo una o dos veces, pero ocasionalmente una inserción dará lugar a una cascada de «acarreos» y el paso (iii) se ejecutará muchas veces. En lo tocante al contador binario, sin embargo, se puede efectuar el trabajo requerido en un tiempo $O(1)$, siempre y cuando nos conformemos con utilizar costes amortizados; véase la Sección 4.6.

Para explicarlo, usaremos un truco de contabilidad según se describe en la sección 4.6. Específicamente, establecemos inicialmente una cuenta bancaria virtual que contiene cero fichas. Cada vez que se crea un nuevo árbol binomial, se cobra al proceso que lo crea una ficha, que se deposita en la cuenta bancaria. Suponemos que basta una ficha para pagar la operación de enlazado, que tal como vimos requiere un tiempo que está en $O(1)$, más un poquito más para pagar un pequeño gasto adicional constante. Siempre que efectuemos la operación de enlazado, retiraremos una ficha de la cuenta para pagarla, así que la operación tiene un coste amortizado nulo. Nuestra cuenta bancaria nunca puede quedar en descubierto, porque nunca se pueden enlazar más árboles que los que hayamos creado. ¿Cuál es el efecto sobre las operaciones de montículos binomiales?

Para la operación de búsqueda del elemento más grande, este truco de contabilidad carece totalmente de efecto: no se crea ni se destruye ningún árbol binomial. En las operaciones de fusión, hay una parte que implica el enlazado de árboles. Más allá de esto, no se crean ni destruyen árboles. Tal como vimos, el enlazado se puede hacer con un coste amortizado nulo, salvo pequeños gastos adicionales. Sin embargo, la operación de fusión puede implicar unir las raíces de los árboles que constituyeran los dos montículos originales, y para las cuales no se ha producido un enlazado. De éstos puede haber $\lceil \log n \rceil$ como máximo. El trabajo requerido para cada uno de ellos se encuentra en $O(1)$, sólo hay que cambiar unos pocos punteros, así que el tiempo total requerido está en $O(\log n)$. Por tanto, una operación de fusión sigue teniendo un coste amortizado que está en $O(\log n)$, a pesar del hecho de que ya se han pagado las operaciones de enlazado necesarias.

Para insertar un elemento adicional en un montículo binomial, comenzaremos en el paso (i) por crear un nuevo árbol binomial. Cuando se crea este árbol, cobramos una ficha que se deposita en la cuenta bancaria. El resto de este paso requiere claramente un tiempo que se encuentra en $O(1)$. De esta manera, aun cuando incluyamos el coste adicional, el paso se puede efectuar en un tiempo que se encuentra en $O(1)$. El paso (ii) es trivial. Cada vez que se ejecuta el paso (iii), se efectúa una operación de enlazado y se ajusta un pequeño número de punteros. Dado que desaparece un árbol binomial durante la operación de enlazado, podemos utilizar una ficha de la cuenta bancaria para pagar todo el paso. El coste amortizado de cada ejecución del paso (iii) es, por tanto, cero. El paso (iv) requiere un tiempo que está en $O(1)$. Resumiendo, se ve que la operación de inserción completa se puede efectuar con un coste amortizado de $O(1)$, tal como se aseguraba al principio de esta sección.

Se han propuesto estructuras de datos aún más elaboradas para implementar las listas dinámicas de prioridad. Empleando *montículos binarios perezosos*, que son iguales que los montículos binomiales salvo que se retrasan algunas operaciones de mantenimiento hasta que sean absolutamente necesarias, se pueden fusionar dos montículos en un tiempo amortizado que está en $O(1)$. El *montículo de Fibonacci* es otra estructura de datos que nos permite fusionar listas de prioridad en un tiempo amortizado constante. Veremos en el problema 6.17 lo útil que puede ser. Estos montículos están basados en *árboles de Fibonacci*; véase el problema 5.29. La estructura denominada montículo de dos extremos, o *montículo doble*, permite encontrar eficientemente tanto el mayor como el menor de los elementos de un conjunto.

5.9 ESTRUCTURAS DE CONJUNTOS DISJUNTOS (PARTICIÓN)

Supongamos que se tienen N objetos numerados de 1 a N. Deseamos agrupar estos objetos en conjuntos disjuntos, de tal modo que en todo momento cada objeto se encuentre exactamente en un conjunto. En cada conjunto, seleccionamos un miembro que servirá como *rótulo* para ese conjunto. Por ejemplo, si decidimos utilizar el más pequeños de los objetos como rótulo, podemos simplemente hacer alusión al conjunto {2, 5, 7, 10} en la forma «conjunto 2». Inicialmente, los N objetos se encuentran en N conjuntos diferentes, cada uno de los cuales contiene exactamente un objeto. En lo sucesivo, ejecutaremos una secuencia de operaciones de dos clases:

dado un cierto objeto, *buscar* el conjunto que lo contiene, y proporcionamos el rótulo de este conjunto; y
dados dos rótulos diferentes, *fusionar* el contenido de los dos conjuntos correspondientes, y seleccionar un rótulo para el conjunto combinado.

Nuestro problema consiste en representar esta situación de forma eficiente en una computadora.

Una posible representación es evidente. Supongamos, tal como se sugiere más arriba, que decidimos utilizar el menor de los elementos de cada conjunto como rótulo. Si declaramos una matriz *conjunto*[1..N], basta con poner el rótulo del con-

junto al que pertenece a cada objeto en el elemento de la matriz que corresponde a dicho objeto. Las dos operaciones que deseamos llevar a cabo se pueden implementar mediante las operaciones siguientes:

```
función buscar1(x)
    {Busca el rótulo del conjunto que contiene x}
    devolver conjunto[x]
procedimiento fusionar1(a, b)
    {Fusiona los conjuntos rotulados como a y b; suponemos que es a ≠ b }
    i ← mín(a, b)
    j ← máx(a, b)
    para k ← 1 hasta N hacer
        si conjunto[k] = j entonces conjunto[k] ← i
```

Supongamos que vamos a ejecutar una secuencia arbitraria de operaciones de los tipos *buscar* y *fusionar*, comenzando a partir de la situación inicial. No sabemos precisamente el orden en que se van a producir estas operaciones. Sin embargo, habrá n del tipo *buscar*, y no habrá más de $N-1$ del tipo *fusionar*, porque al cabo de $N-1$ operaciones *fusionar* todos los objetos están en el mismo conjunto. Además, en muchas aplicaciones, n es comparable con N. Si la consulta o modificación de un elemento cuenta como una operación elemental, entonces está claro que *buscar*1 requiere un tiempo constante, y que *fusionar*1 requiere un tiempo que está en $\Theta(N)$. Las n operaciones *buscar*, por tanto, requieren un tiempo en $\Theta(n)$, mientras que las $N-1$ operaciones *fusionar* requieren un tiempo que está en $\Theta(N^2)$. Si n y N son comparables, entonces toda la secuencia de operaciones requiere un tiempo que se encuentra en $\Theta(n^2)$.

Intentemos hacerlo mejor. Sin dejar de utilizar una sola matriz, podemos representar cada conjunto como un árbol con raíz, en el cual cada nodo contiene un único puntero que apunta a su padre (tal como en el tipo *nodoárbol*2 de la Sección 5.5). Adoptamos el convenio siguiente: si *conjunto*[i] = i, entonces i es a la vez el rótulo de su conjunto y la raíz del árbol correspondiente; si *conjunto*[i] = $j \neq i$, entonces j es el padre de i en algún árbol. La matriz

1	2	3	2	1	3	4	3	3	4

representa por tanto los árboles mostrados en la figura 5.21, que a su vez representa los conjuntos {1, 5}, {2, 4, 7, 10} y {3, 6, 8, 9}. Para fusionar dos conjuntos, ahora se necesita cambiar solamente un valor de la matriz; por otra parte, es más difícil hallar el conjunto al que pertenece el elemento.

```
función buscar2(x)
    {Busca el rótulo del conjunto que contiene al elemento x}
    r ← x
    mientras conjunto[r] ≠ r hacer r ← conjunto[r]
    devolver r
```

procedimiento *fusionar2*(*a*, *b*)
{Fusiona los conjuntos cuyos rótulos son *a* y *b*; suponemos $a \neq b$}
si $a < b$ **entonces** *conjunto*[*b*] $\leftarrow a$
sino *conjunto*[*a*] $\leftarrow b$

Ahora si cada consulta o modificación de un elemento de una matriz cuenta como una operación elemental, entonces el tiempo necesario para ejecutar una operación *fusionar* es constante. Sin embargo, el tiempo necesario para ejecutar una operación *buscar* está en $\Theta(N)$ en el caso peor. Cuando las cosas van mal, por tanto, la ejecución de una secuencia arbitraria de n operaciones *buscar*2 y de $N-1$ operaciones *fusionar*2 a partir de la situación inicial puede requerir un tiempo que esté en $\Theta(nN)$. Si n es comparable a N, según se suponía anteriormente, esto es $\Theta(n^2)$ y no hemos ganado nada con respecto al uso de *buscar*1 y *fusionar*1. El problema surge porque después de k llamadas a *fusionar*2 podemos enfrentarnos a un árbol de altura k, así que toda llamada subsiguiente a *buscar*2 puede requerir un tiempo proporcional a k. Para evitar esto, debemos encontrar una forma de limitar la altura de los árboles producidos.

Hasta el momento, hemos decidido arbitrariamente utilizar como rótulo el miembro más pequeño de los conjuntos. Esto significa que cuando se fusionan dos árboles, el que *no* contiene el miembro más pequeño del conjunto combinado resultante pasa a ser un sub-árbol de aquél que lo contiene. Sería mejor disponer las cosas de tal forma que siempre fuera el árbol cuya altura fuera menor el que pasara a ser un sub-árbol del otro. Supongamos que tenemos que fusionar dos árboles cuyas alturas son, respectivamente, h_1 y h_2. Empleando esta técnica, la altura del árbol fusionado resultante será máx(h_1, h_2) si $h_1 \neq h_2$, o bien $h_1 + 1$ si $h_1 = h_2$. Tal como muestra el siguiente teorema, de esta manera la altura de los árboles no crece tan rápidamente.

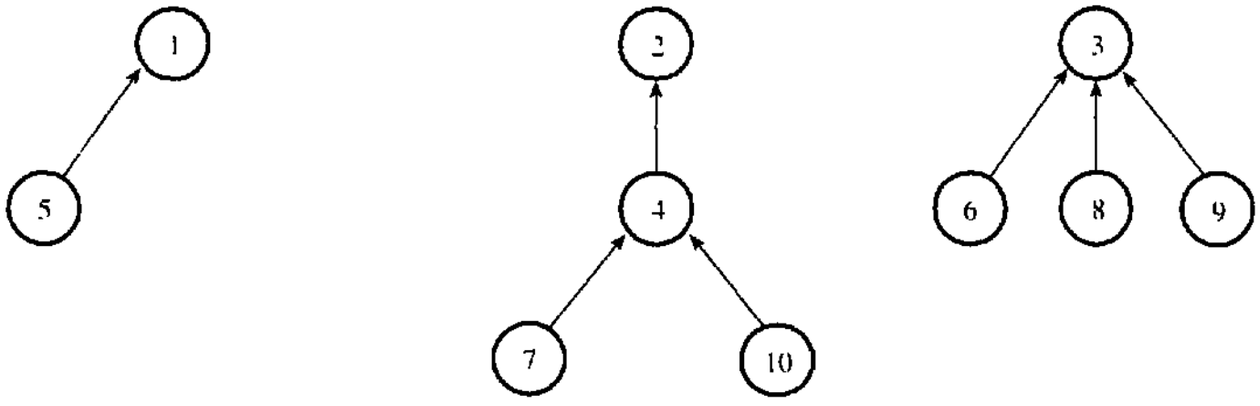


Figura 5.21. Representación de conjuntos disjuntos en forma de árbol

Teorema 5.9.1 *Empleando la técnica esbozada anteriormente, al cabo de una secuencia arbitraria de operaciones de fusión que comienzan en la situación inicial, un árbol que contenga* k *nodos tiene una altura que es como máximo* $\lfloor \lg k \rfloor$.

Demostración. La demostración se hace por inducción matemática generalizada sobre k, el número de nodos que hay en el árbol:

◊ *Base*: El teorema es claramente cierto cuando $k = 1$, porque un árbol con un solo nodo tiene altura 0, y $0 \leq \lfloor \lg 1 \rfloor$.

◊ *Paso de inducción*: Considérese cualquier $k > 1$. Considérese como hipótesis de inducción que el teorema es cierto para todos los m tales que $1 \leq m < k$. Un árbol que contenga k nodos se puede obtener solamente por fusión de dos árboles más pequeños. Supongamos que estos dos árboles más pequeños contienen respectivamente a y b nodos, en donde se puede suponer sin pérdida de generalidad que es $a \leq b$. Ahora bien, $a \geq 1$ puesto que no hay forma de obtener un árbol de 0 nodos partiendo de la situación inicial, y $k = a + b$. Se sigue que es $a \leq k/2$ y $b \leq k-1$. Puesto que $k > 1$, será $k/2 < k$ y $k-1 < k$, y por tanto $a < k$ y $b < k$. Sean h_a y h_b respectivamente las alturas de estos dos árboles menores, y sea h_k la altura del árbol fusionado resultante. Se pueden dar dos casos.

Si $h_a \neq h_b$, entonces $h_k = \max(h_a, h_b) \leq \max(\lceil \lg a \rceil, \lceil \lg b \rceil)$, donde hemos utilizado la hipótesis de inducción dos veces para obtener la desigualdad. Dado que tanto a como b son menores que k, se sigue que es $h_k \leq \lfloor \lg k \rfloor$.

Si $h_a = h_b$, entonces $h_k = h_a + 1 \leq \lceil \lg a \rceil + 1$, utilizando una vez más la hipótesis de inducción. Ahora $\lfloor \lg a \rfloor \leq \lfloor \lg(k/2) \rfloor = \lfloor \lg(k) - 1 \rfloor = \lfloor \lg k \rfloor - 1$ y por tanto $h_k \leq \lfloor \lg k \rfloor$.

Entonces el teorema es cierto cuando $k = 1$, y su veracidad para $k = n > 1$ se sigue de la suposición de su veracidad para $k = 1, 2, \ldots, n-1$. Por el principio de inducción matemática generalizada, el teorema es por tanto cierto para todos los $k \geq 1$.

La altura de los árboles se puede mantener en una matriz adicional *altura*[1..*N*], de tal modo que *altura*[*i*] proporcione la altura del nodo *i* en su árbol actual. Cuando *a* es el rótulo de un conjunto, *altura*[*a*] nos da consiguientemente la altura del árbol correspondiente, y de hecho éstas son las únicas alturas que nos conciernen. Inicialmente, *altura*[*i*] se pone a cero para todo *i*. El procedimiento *buscar*2 queda intacto, pero es preciso modificar *fusionar* adecuadamente:

```
procedimiento fusionar3(a, b)
    {Fusiona los conjuntos cuyos rótulos son a y b; suponemos que a ≠ b}
    si altura[a] = altura[b]
    entonces
        altura[a] ← altura[a] + 1
        conjunto[b] ← a
    sino
        si altura[a] > altura[b]
        entonces conjunto[b] ← a
        sino conjunto[a] ← b
```

Si cada consulta o modificación de un elemento de la matriz cuenta como una operación elemental, el tiempo necesario para ejecutar una secuencia arbitraria de n operaciones *buscar*2 y $N-1$ operaciones *fusionar*3, comenzando a partir de la si-

tuación inicial, es $\Theta(N + n \log N)$ en el caso peor; suponiendo que n y N sean comparables, este tiempo está en el orden exacto de $n \log n$.

Modificando *buscar*2, podemos hacer que nuestras operaciones sean todavía más rápidas. Cuando se está intentando determinar el conjunto que contiene un cierto elemento x, primero se recorren las aristas del árbol que suben desde x hacia la raíz. Una vez que se conoce la raíz, podemos recorrer una vez más las mismas aristas, modificando esta vez cada nodo encontrado por el camino de tal manera que su puntero señale ahora directamente la raíz. Esta técnica se denomina *compresión de caminos*. Por ejemplo, cuando ejecutamos la operación *buscar*(20) en el árbol de la figura 5.22a, el resultado es el árbol de la figura 5.22b: los nodos 20, 10 y 9, que se encontraban en el camino que va desde el nodo 20 hasta la raíz, apuntan ahora directamente a la raíz. Los punteros de los nodos restantes no se han modificado.

Esta técnica tiende evidentemente a reducir la altura del árbol y por tanto acelera las operaciones *buscar* subsiguientes. Por otra parte, la nueva operación *buscar* recorre dos veces el camino que va desde el nodo en cuestión hasta la raíz, y por tanto requiere aproximadamente el doble de tiempo. Por tanto, la compresión de caminos puede no merecer la pena si sólo se efectúa un pequeño número de operaciones *buscar*. Sin embargo, si se efectúa un gran número de operaciones *buscar*, entonces cabe esperar que al cabo de un tiempo, hablando en términos generales, todos los nodos implicados queden asociados directamente con las raíces de sus respectivos árboles, así que las operaciones *buscar* subsiguientes requerirán un tiempo constante. Una operación *fusionar* perturbará esta situación sólo de forma ligera, y no durante mucho tiempo. La mayor parte del tiempo, por tanto, tanto las operaciones *buscar* como las operaciones *fusionar* requerirán un tiempo en $O(1)$, así que una secuencia de n de las primeras y de N–1 de las segundas operaciones requerirá un tiempo que va a estar (casi) en $O(n)$ cuando n y N sean comparables. Véase más adelante lo cerca que estamos de esta situación ideal.

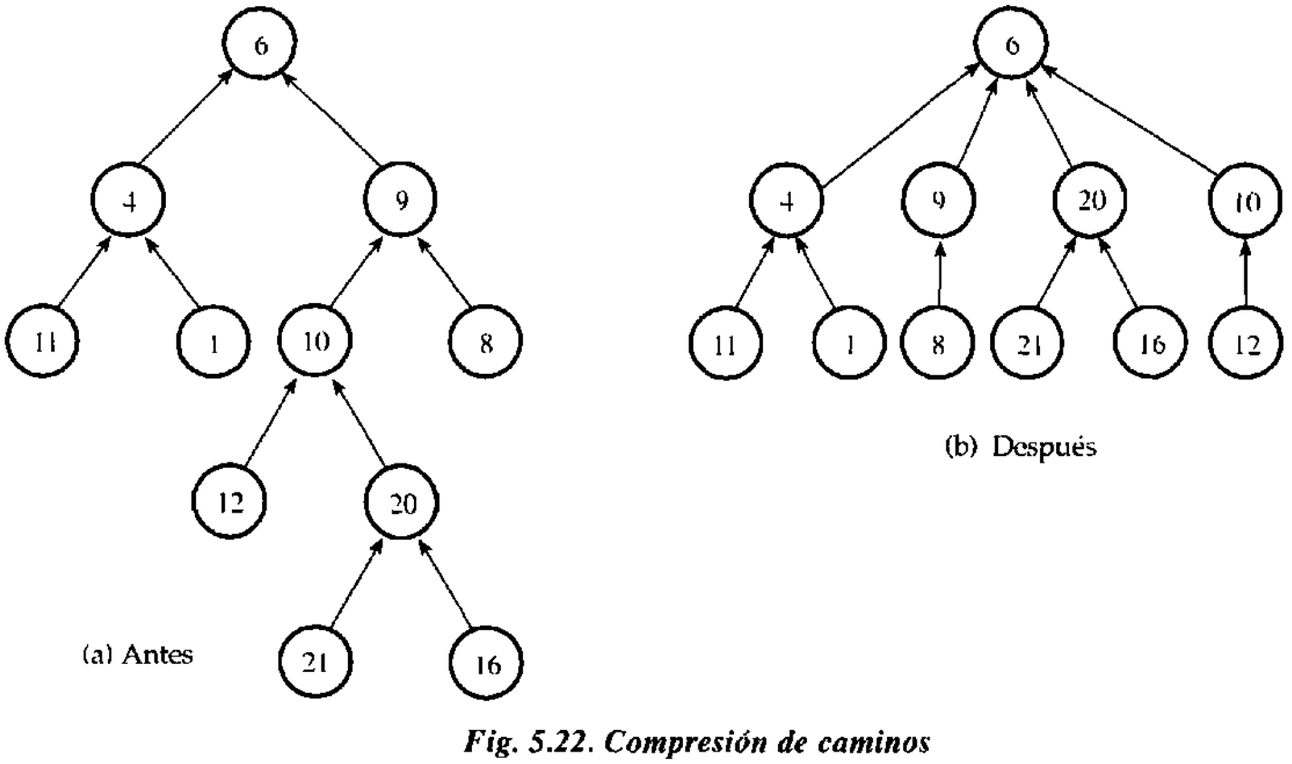


Fig. 5.22. Compresión de caminos

Queda por aclarar un pequeño asunto. Utilizando la compresión de caminos, ya no es cierto que la altura de un árbol cuya raíz sea *a* esté dada por *altura*[*a*]. Esto se debe a que la compresión de caminos puede modificar la altura de un árbol, pero no afecta al contenido del nodo raíz. Sin embargo, la compresión de caminos puede reducir la altura de un árbol, pero nunca puede incrementarla, así que si *a* es la raíz, entonces sigue siendo cierto que *altura*[*a*] es una cota superior de la altura del árbol; véase el problema 5.31. Para evitar confusiones, llamaremos a este valor el *rango* del árbol; el nombre de la matriz utilizada en *fusionar*3 debería de modificarse en consonancia. La función *buscar* queda ahora como sigue:

```
función buscar3(x)
    {Busca el rótulo del conjunto que contiene el elemento x}
    r ← x
    mientras conjunto[r] ≠ r hacer r ← conjunto[r]
    { r es la raíz del árbol }
    i ← x
    mientras i ≠ r hacer
        j ← conjunto[i]
        conjunto[i] ← r
        i ← j
    devolver r
```

Desde ahora en adelante, cuando utilicemos esta combinación de dos matrices y los procedimientos *buscar*3 y *fusionar*3 para tratar conjuntos disjuntos de objetos, diremos que estamos utilizando una *estructura de partición,* véanse también los problemas 5.32 y 5.33 para otras variaciones sobre este mismo tema.

No resulta fácil analizar el tiempo necesario para una secuencia arbitraria de operaciones *buscar* y *fusionar* cuando se utiliza la compresión de caminos. Primero es necesario definir dos nuevas funciones $A(i, j)$ y $\alpha(i, j)$.

La función $A(i, j)$ es una pequeña variación de la *función de Ackermann*; véase el problema 5.38. Está definida para $i \geq 0$ y $j \geq 1$:

$$A(i,j) = \begin{cases} 2j & \text{si } i = 0 \\ 2 & \text{si } j = 1 \\ A(i-1, A(i, j-1)) & \text{en caso contrario} \end{cases}$$

Los hechos siguientes se siguen fácilmente desde la definición:

◊ $A(1,1) = 2$ y $A(1,j+1) = A(0, A(2,j))$ para $j \geq 1$. Se sigue que $A(1,j) = 2j$ para todo j.

◊ $A(2,1) = 2$ y $A(2,j+1) = A(1,A(2,j)) = 2^{A(2,j)}$ para $j \geq 1$. Por tanto, $A(2,1) = 2$, $A(2,3) = 2^{A(2,2)} = 2^{2^2} = 16$, $A(2,4) = 2^{A(2,3)} = 2^{2^{2^2}} = 65536$ y en general

$$A(2, j) = 2 \cdot \left.\begin{matrix} \\ \\ \end{matrix}\right\} j \text{ doses}$$

en donde el lado derecho contiene j doses en conjunto. Recuerde que la exponenciación es asociativa por la derecha, así que

$$2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65536$$

◊ $A(3, 1) = 2$, $A(3, 2) = A(2, A(3, 1)) = 4$
$A(3, 3) = A(2, A(3, 2)) = 65536$

$A(3, 4) = A(2, A(3, 3)) = A(2, 65536) = 2 \cdot \left.\begin{matrix} \\ \\ \end{matrix}\right\}$ 65536 doses, y así sucesivamente.

Queda claro que la función A crece extremadamente deprisa.
Ahora bien, la función $\alpha(i, j)$ se define como una especie de inversa de A:

$$\alpha(i, j) = \text{mín}\{k \mid k \geq 1 \text{ y } A(k, 4\lceil i / j \rceil) > \lg j\}.$$

Del mismo modo que A crece muy rápidamente, α crece con extremada lentitud. Para ver esto, obsérvese que para todo j fijo, $\alpha(i, j)$ se maximiza cuando $i \leq j$, en cuyo caso es $4\lceil i/j \rceil = 4$, así que

$$\alpha(i, j) \leq \text{mín}\{k \mid k \geq 1 \text{ y } A(k, 4) > \lg j\}.$$

Por tanto $\alpha(i, j) > 3$ solamente en el caso de que

$$\lg j \geq A(3, 4) = 2 \cdot \left.\begin{matrix} \\ \\ \end{matrix}\right\} 65536 \text{ doses}$$

que es un número enorme. Por tanto, salvo para valores astronómicos de j, $\alpha(i, j) \leq 3$.

Con un universo de N objetos y la situación inicial dada, considérese una secuencia arbitraria de n llamadas a *buscar*3 y $m \leq N-1$ llamadas a *fusionar*3. Sea $c = n + m$. Utilizando las funciones anteriores, Tarjan consiguió mostrar que una tal secuencia se puede ejecutar en un tiempo que está en $\Theta(c\alpha(c,N))$ en el caso peor. (Por supuesto, seguimos suponiendo que toda consulta o modificación de un elemento de la matriz cuenta como una operación elemental.) Dado que para todos los efectos prácticos podemos suponer que $\alpha(c, N) \leq 3$, el tiempo requerido por la secuencia de operaciones es esencialmente lineal en c. Sin embargo, ningún algoritmo conocido nos permite efectuar la secuencia en un tiempo que sea *verdaderamente* lineal en c.

5.10 PROBLEMAS

Problema 5.1. Tiene usted que implementar una pila de elementos de un tipo dado. Dar las declaraciones necesarias, y escribir tres procedimientos para inicializar la pila, añadir un nuevo elemento y eliminar un elemento, respectivamente. Incluya en los procedimientos las comprobaciones necesarias para evitar añadir o eliminar demasiados elementos. Escriba también una función que proporcione el elemento que se encuentre en ese momento en la parte superior de la pila. Asegúrese de que esta función se comporte de forma sensata cuando la pila esté vacía.

Problema 5.2. Se puede representar una cola empleando una matriz de elementos del tipo necesario, junto con dos punteros. Uno de ellos da el índice del elemento que se encuentra en la cabeza de la cola (el próximo en salir), y el otro da el índice del elemento que se encuentra al final de la cola (el último en llegar). Dar las declaraciones necesarias, y escribir tres procedimientos para inicializar la cola, añadir un elemento nuevo y eliminar un elemento, respectivamente. Si la matriz contiene espacio para n elementos, ¿cuál es el máximo número de elementos que pueden estar en la cola? ¿Cómo sabe cuándo está vacía y cuándo está llena? Incluya en los procedimientos las comprobaciones que eviten añadir o eliminar demasiados elementos. Escriba además una función que proporcione el elemento que se encuentre en ese momento a la cabeza de la cola. Asegúrese de que esta función se comporte de forma sensata cuando la cola esté vacía.

Pista: A medida que se añaden o eliminan elementos, la cola se desplaza a lo largo de la matriz. Cuando un índice se sale de la matriz, no copie todos los elementos en una nueva posición: deje que el puntero vuelva a empezar por el otro extremo de la cola.

Problema 5.3. Completar los detalles de la técnica denominada iniciación virtual que se describía en la Sección 5.1 y está ilustrada en la figura 5.1. Hay que escribir tres algoritmos:

```
procedimiento iniciar
    {Inicializa virtualmente T[1..N]}

procedimiento almacenar(i, v)
    {Da a T[i] el valor v}

función val(i)
    {Proporciona el valor de T[i] si es que
    ha recibido un valor;
    proporciona un valor por omisión
    (tal como –1) en caso contrario}
```

Las llamadas a cualquiera de ellos (¡incluyendo *iniciar*!) deben requerir un tiempo constante en el caso peor.

Problema 5.4. ¿En qué medida cambia su solución del Problema 5.3 si el índice de la matriz T, en lugar de ir desde 1 hasta n, va (digamos) desde n_1 hasta n_2?

Problema 5.5. Sin escribir los algoritmos detallados, esboce la forma de adaptar la iniciación virtual a una matriz bidimensional.

Problema 5.6. Muestre con cierto detalle la forma en que se podría representar en una máquina el grafo dirigido de la figura 5.3, empleando (a) el tipo de representación *grafoadya*, y (b) el tipo de representación *listagraf*.

Problema 5.7. Dibuje los tres árboles distintos que hay con cinco nodos, y los seis árboles distintos que hay con seis nodos. Repita el problema para los árboles con raíz. Para este problema, el orden de las ramas de un árbol con raíz carece de importancia. Debería encontrar nueve árboles con raíz con cinco nodos, y veinte con seis nodos.

Problema 5.8. Continuando con el Problema 5.7, ¿cuántos árboles con raíz hay con seis nodos si se tiene en cuenta el *orden*?

Problema 5.9. Mostrar la forma en que se representarían los cuatros árboles con raíz que se ilustran en la figura 5.4b empleando punteros desde cada nodo hasta su hijo mayor y hasta su hermano siguiente.

Problema 5.10. A costa de un *bit* de espacio adicional en los registros de tipo *nodoárbol*1, podemos hacer posible hallar el padre de cualquier

nodo dado. La idea es utilizar el campo *hermano siguiente* del miembro situado más a la derecha en cualquier conjunto de hermanos para apuntar a su padre común. Dar los detalles de esta aproximación. En un árbol que contiene n nodos, ¿cuánto tiempo se necesita para hallar el padre de un nodo dado en el caso peor? ¿Cambia su respuesta si se sabe que un nodo del árbol nunca puede tener más de k hijos?

Problema 5.11. Reescribir el algoritmo *buscar* de la Sección 5.5 para evitar las llamadas recursivas.

Problema 5.12. Dar una implementación detallada de las tablas asociativas empleando listas del tipo *lista* de la Sección 5.6. Su implementación debe proporcionar los algoritmos siguientes.

función *inicializar*(T, x)
función *val*(T, x)
función *modif*(T, x, y)

Estos algoritmos determinan si $T[x]$ ha recibido valor inicial, acceden a su valor actual si lo tiene, y dan a $T[x]$ el valor y (bien creando la entrada $T[x]$ o modificando su valor si ya existe), respectivamente.

Problema 5.13. Demostrar que una secuencia de n accesos a una tabla asociativa implementada en la forma expuesta en el problema 5.12 requiere un tiempo que se encuentra en $\Omega(n^2)$ en el caso peor.

Problema 5.14. Demostrar que incluso en el caso de que la matriz utilizada para implementar una tabla de dispersión tenga un tamaño $N = m^2$, en donde m es el número de elementos que hay que almacenar en la tabla, la probabilidad de colisión es significativa. Suponga que la función de dispersión envía cada elemento a una posición aleatoria dentro de la matriz.

Problema 5.15. Demostrar que el coste de una redistribución se puede despreciar incluso en el caso peor, siempre y cuando efectuemos un análisis amortizado. En otras palabras, mostrar que los accesos a la tabla pueden poner en la cuenta bancaria un número de fichas suficiente para pagar el coste completo de la redistribución cada vez que el factor de carga sobrepase la unidad. Suponga que tanto la selección de una nueva función de dispersión como la redistribución de una entrada en la tabla requieren un tiempo constante.

Problema 5.16. Demostrar que si se utiliza la dispersión para implementar la tabla de símbolos de un compilador, si se mantiene por debajo de 1 el factor de carga, y si se hace la suposición poco natural consistente en que todo posible identificador tiene igual probabilidad de ser utilizado, entonces la probabilidad de que cualquier identificador colisione con más de t identificadores distintos es menor que $1/t$, para cualquier entero t. Concluir que el tiempo medio requerido para una secuencia de accesos a la tabla está en $O(n)$.

Problema 5.17. Proponga estrategias distintas del encadenamiento para gestionar las colisiones en una tabla de dispersión.

Problema 5.18. Dibujar un árbol binario esencialmente completo que tenga (a) 15 nodos y (b) 16 nodos.

Problema 5.19. En la Sección 5.7 se vio un algoritmo para crear un montículo (*crear-montículo-lento*) que describíamos como «bastante ineficiente». Analizar el caso peor para este algoritmo, y compararlo con el algoritmo de tiempo lineal *crear-montículo*.

Problema 5.20. Sea $T[1..12]$ una matriz tal que $T[i] = i$ para todo $i \leq 12$. Mostrar el estado de la matriz al cabo de las siguientes llamadas a procedimiento. Las llamadas se

efectúan una tras otra, y todas ellas salvo la primera operan sobre la matriz que dejara su predecesor.

crear-montículo(*T*)
modif-montículo(*T*, 12, 10)
modif-montículo(*T*, 1, 6)
modif-montículo(*T*, 5, 8)

Problema 5.21. Mostrar un montículo T que contenga n valores diferentes, de tal manera que la secuencia siguiente dé lugar a un montículo distinto.

$m \leftarrow$ *máximo*($T[1..n]$)
borrar-máximo($T[1..n]$)
añadir($T[1..n]$, m)

Dibuje el montículo después de cada operación. Puede seleccionar el n que más le guste.

Problema 5.22. Diseñar un algoritmo *buscar-en-montículo*($T[1..n]$, x) que busque el valor x en el montículo T, y que proporcione el índice de x en el montículo, si está presente, o bien 0 en caso contrario. ¿Cuál es el tiempo de ejecución de su algoritmo? ¿Puede utilizar la propiedad del montículo para acelerar la búsqueda, y, de ser así, cómo?

Problema 5.23 (montículos *k*-arios). En la Sección 5.7 definíamos los montículos en términos de un *árbol binario* esencialmente completo. Debería estar claro que la idea se puede generalizar a los árboles k-arios esencialmente completos, para todo $k \geq 2$. Demuestra que se pueden hacer corresponder los nodos de un árbol k-ario que contenga n nodos con los elementos $T[0]$ a $T[n-1]$ de una matriz, y esto de tal forma que el padre del nodo representado en $T[i]$ se encuentre en $T[(i-1) \div k]$ para $i > 0$, y de tal modo que los hijos del nodo representado en $T[i]$ se encuentren en $T[ik+1]$, $T[ik+2]$,..., $T[(i+1)k]$. Observe que ésta *no* es la correspondencia que empleamos para los árboles binarios descritos en la Sección 5.7; allí se empleaba una correspondencia con $T[1..n]$ y no con $T[0..n-1]$.

Escriba los procedimientos *hundir*(T, k, i) y *flotar*(T, k, i) para estos montículos generalizados. ¿Cuáles son las ventajas y desventajas de estos montículos generalizados? Para ver una aplicación en la cual pueden resultar útiles, véase el Problema 6.16.

Problema 5.24. Para *ordenar por montículo*, ¿cuáles son las mejores y peores disposiciones iniciales de los elementos que hay que ordenar, en lo tocante al tiempo de ejecución del algoritmo? Justifique su respuesta.

Problema 5.25. Demostrar que el montículo binomial B_i definido en la Sección 5.8 contiene 2^i nodos, de los cuales $\binom{i}{k}$ se encuentran en la profundidad k, con $0 \leq k \leq i$.

Problema 5.26. Demostrar que un montículo binomial que contenga n elementos consta como máximo de $\lceil \lg n \rceil$ árboles binomiales, el mayor de los cuales contiene $2^{\lceil \lg n \rceil}$ elementos.

Problema 5.27. Considere el algoritmo para insertar un elemento nuevo en un montículo binomial H, que se daba en la Sección 5.8. Un método más sencillo consistiría en crear un árbol binomial B_0^* tal como en el paso (i) del algoritmo, transformar ese árbol en un montículo binomial y fusionar con H este nuevo montículo. ¿Por qué hemos preferido el algoritmo más complicado?

Problema 5.28. Utilizando el truco de contabilidad que se describe en la Sección 5.8, ¿cuál es el coste amortizado de borrar el elemento más grande de un montículo binomial?

Problema 5.29 (árboles de Fibonacci). Resulta conveniente definir el árbol de Fibo-

nacci F_{-1} como formado por un solo nodo. Entonces el i-ésimo árbol de Fibonacci, F_i, $i \geq 0$, se define recursivamente como formado por un nodo raíz con i hijos, en donde el j-ésimo hijo, $1 \leq j \leq i$, es a su vez la raíz de un árbol de Fibonacci F_{i-2}. La figura 5.23 muestra desde F_0 hasta F_4. Demostrar que el árbol de Fibonacci F_i, $i \geq 0$, tiene f_{i+1} nodos, en donde f_k es el k-ésimo miembro de la sucesión de Fibonacci; véase la Sección 1.6.4.

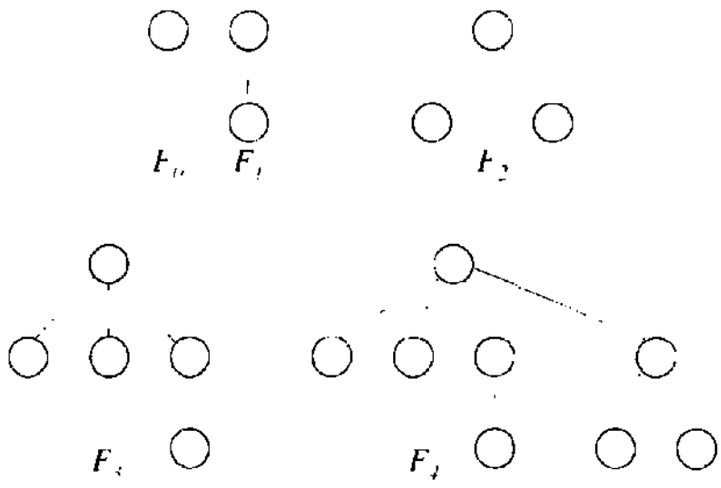


Figura 5.23. Los árboles de Fibonacci de F_0 a F_4

Problema 5.30. Si toda consulta o modificación de un elemento de una matriz cuenta como una operación elemental, demostrar que el tiempo necesario para ejecutar una secuencia arbitraria de n operaciones del tipo *buscar*2 y N–1 operaciones del tipo *fusionar*3 partiendo de la situación inicial está en $\Theta(N + n \log N)$; esto es $\Theta(n \log n)$ cuando n y N son comparables.

Problema 5.31. Cuando se utiliza la compresión de caminos, nos contentamos con utilizar una matriz *rango* que nos da una cota superior de la altura del árbol, en lugar de la altura exacta. Estime el tiempo que se necesitaría para recalcular la altura exacta del árbol después de cada compresión de un camino.

Problema 5.32. En la Sección 5.9 se describía la fusión de dos árboles de tal forma que el árbol cuya altura sea menor pasa a ser un sub-árbol del otro. Una segunda táctica posible consiste en asegurar que el árbol que contenga el menor número de nodos pase siempre a ser un sub-árbol del otro. La compresión de caminos no modifica el número de nodos que hay en un árbol, así que es fácil almacenar este valor exactamente, mientras que no se podría llevar eficientemente la cuenta de la altura exacta del árbol después de la compresión de caminos. Escriba un procedimiento *fusionar*4 para implementar esta táctica y demuestre un resultado correspondiente al teorema 5.9.1.

Problema 5.33. La raíz de un árbol no tiene padre, y nunca utilizamos el valor de *rango* para un nodo que no sea una raíz. Utilice esta propiedad para implementar una estructura de partición sin más que una matriz de longitud N en lugar de dos (*conjunto* y *rango*).

Problema 5.34. Sea A la variante de la función de Ackermann que se describe en la Sección 5.9. Mostrar que es $A(2, i) = 4$ para todo i.

Problema 5.35. Sea A la variante de la función de Ackermann que se describe en la Sección 5.9. Demostrar que $A(i + 1, j) \geq A(i, j)$ y que $A(i, j + 1) \geq A(i, j)$ para todo i, j.

Problema 5.36. Sea A la variante de la función de Ackermann que se describe en la Sección 5.9, y definamos $a(i, n)$ en la forma

$$a(i, n) = \min\{j \mid A(i, j) > \lg n\}.$$

Demostrar que $a(1, n)$ está en $O(\log \log n)$.

Problema 5.37. Considere la función $\lg^* (n)$, que es el *logaritmo iterado* de n. Informalmente, se trata del número de veces que hay que aplicar la función lg a n para obtener un valor menor o igual que 1. Por ejemplo, $\lg 16 = 4$, $\lg 4 = 2$ y $\lg 2 = 1$; por tanto $\lg \lg \lg 16 = 1$, y así $\lg^* 16 = 3$. De forma algo más formal, definimos $\lg^* n$ en la forma

$$\lg^* n = \min\{k \mid \underbrace{\lg \lg \ldots \lg}_{k \text{ veces}} n \leq 1\}.$$

La función $\lg^* n$ crece muy lentamente: $\lg^* n$ es 4 ó menos para todo $n < 65536$. Sea $a(i, n)$ la función definida en el problema anterior. Mostrar que $a(2, n)$ está en $O(\lg^* n)$.

Problema 5.38. La función de Ackermann (la verdadera, en este caso) se define en la forma

$$A(i,j) = \begin{cases} j+1 & \text{si } i = 0 \\ A(i-1,1) & \text{si } i > 0, j = 0 \\ A(i-1, A(i, j-1)) & \text{en caso contrario.} \end{cases}$$

Calcular $A(2, 5)$, $A(3, 3)$ y $A(4, 4)$.

5.11 REFERENCIAS Y TEXTOS MÁS AVANZADOS

Para más información acerca de las estructuras de datos, consulte Knuth (1968, 1973), Stone (1972), Horowitz y Sahni (1976), Standish (1980), Aho, Hopcroft y Ullman (1983), Tarjan (1983), Gonnet y Baeza-Yates (1984), Kingston (1990), Lewis y Denenberg (1991) y Wood (1993). Los grafos y los árboles se presentan desde un punto de vista matemático en Berge (1958, 1970).

La inicialización virtual proviene del Ejercicio 2.12 de Aho, Hopcroft y Ullman (1974). Existe toda una gama de métodos para mantener equilibrados los árboles de búsqueda; no los hemos explicado en este capítulo porque no nos harán falta más adelante. Entre éstos, los árboles AVL, que provienen de Adel'son – Vel'skiĭ y Landis (1962), y se describen con detalle en Knuth (1973); los árboles 2-3 vienen de Aho, Hopcroft y Ullman (1974); los árboles rojinegros son de Guibas y Sedgewick (1978), y los árboles biselados, que ofrecen un buen rendimiento en el caso peor en el sentido amortizado, son de Sleator y Tarjan (1985). Lea también Tarjan (1983). Las tablas de dispersión se tratan con detalle en Knuth (1973); allí se dan muchas alternativas distintas a las de las tablas abiertas.

El montículo se presentó por Williams (1964) como estructura de datos para ordenaciones. Los montículos con más de dos hijos por nodo interno son de Johnson (1975, 1977); véanse los Problemas 5.23 y 6.16. La idea de acelerar la *ordenación por el método del montículo* hundiendo los nodos hasta el fondo antes de flotarlos hasta su posición correcta es de Carlsson (1987a). Para ideas acerca de la construcción más rápida de montículos, consulte McDiarmid y Reed (1989). Véase también Gonnet y Munro (1986), y Schaffer y Sedgewick (1993).

Los montículos binomiales son de Vuillemin (1978); véase también Brown (1978) y Kozen (1992). Los montículos de Fibonacci son de Fredman y Tarjan (1987), que es el lugar en que se muestra que se pueden utilizar para acelerar varios algoritmos clásicos de optimización de redes del tipo de los que estudiaremos en el Capítulo 6; véase el Problema 6.17 para un ejemplo. Los montículos de dos extremos se deben a Carlsson (1986, 1987b), que son buenas fuentes de ideas acerca de los montículos en general.

El análisis de estructuras de partición (de conjuntos disjuntos) que implican la función de Ackermann se debe a Tarjan (1975), pero es más fácil de leer la relación subsiguiente en Tarjan (1983). Un cota superior previa para la complejidad de este problema en el caso peor se debía a Hopcroft y Ullman (1973); contenía la función $\lg^*$ del Problema 5.37. Galil e Italiano (1991) revisan una buena cantidad de algoritmos propuestos. En este libro daremos solamente algunas de las posibles aplicaciones de las estructuras de partición; para más aplicaciones, véase Hopcroft y Karp (1971), Aho, Hopcroft y Ullman (1974, 1976), y Nelson y Oppen (1980). La función de Ackermann se debe a Ackermann (1928).

Capítulo 6

Algoritmos voraces[1]

Si los algoritmos voraces son la primera familia de algoritmos que examinaremos con detalle en este libro, la razón es sencilla: suelen ser los más fáciles. Tal como su nombre indica, el enfoque que aplican *es miope*, y toman decisiones basándose en la información que tienen disponible de modo inmediato, sin tener en cuenta los efectos que estas decisiones puedan tener en el futuro. Por tanto resultan *fáciles de inventar*, fáciles de implementar y, cuando funcionan, son eficientes. Sin embargo, como el mundo no suele ser tan sencillo, hay muchos problemas que no se pueden resolver correctamente con un enfoque tan grosero.

Los algoritmos voraces se utilizan típicamente para resolver problemas de optimización. Los ejemplos posteriores de este capítulo incluyen la búsqueda de la ruta más corta para ir desde un nodo a otro a través de una red de trabajo o la búsqueda del mejor orden para ejecutar un conjunto de tareas en una computadora. En tales contextos, un algoritmo voraz funciona seleccionando el arco, o la tarea, que parezca más prometedora en un determinado instante; nunca reconsidera su decisión, sea cual fuere la situación que pudiera surgir más adelante. No hay necesidad de evaluar alternativas, ni de emplear sofisticados procedimientos de seguimiento que permitan deshacer las decisiones anteriores. Comenzaremos este capítulo con un ejemplo cotidiano en el que esta táctica funciona bien.

6.1 DAR LA VUELTA (1)

Supongamos que vivimos en un país en el que están disponibles las siguientes monedas: 100 pesetas, 25 pesetas, 10 pesetas, 5 pesetas y 1 peseta. Nuestro problema consiste en diseñar un algoritmo *para pagar una cierta cantidad a un cliente*, utilizando el menor número posible de monedas. Por ejemplo, si tenemos que pagar 289 pesetas, la mejor solución consiste en dar al cliente 10 monedas: 2 de 100, 3 de 25, 1 de 10 y 4 de 1 peseta. Casi todos nosotros resolvemos este tipo de problema todos los días sin pensarlo dos veces, empleando de forma inconsciente un

algoritmo evidentemente voraz: empezamos por nada, y en cada fase vamos añadiendo a las monedas que ya estén seleccionadas una moneda de la mayor denominación posible, pero que no debe llevarnos más allá de la cantidad que haya que pagar.

El algoritmo se puede formalizar de la manera siguiente:

```
función devolver cambio(n): conjunto de monedas
    {Da el cambio de n unidades utilizando el menor
     número posible de monedas. La constante C
     especifica las monedas disponibles}
    const C = {100, 25, 10, 5, 1}
    S ← Ø {S es un conjunto que contendrá la solución}
    s ← 0 {s es la suma de los elementos de S}
    mientras s ≠ n hacer
        x ← el mayor elemento de C tal que s + x ≤ n
        si no existe ese elemento entonces
            devolver «no encuentro la solución»
        S ← S ∪ {una moneda de valor x}
        s ← s + x
    devolver S
```

Es fácil convencerse (pero sorprendentemente difícil de probar formalmente) de que con los valores dados para las monedas, y disponiendo de un suministro adecuado de cada denominación, este algoritmo siempre produce una solución óptima para nuestro problema. Sin embargo, con una serie de valores diferente, o si el suministro de alguna de las monedas está limitado, el algoritmo voraz puede no funcionar; véanse los problemas 6.2 y 6.4. En algunos casos, puede seleccionar un conjunto de monedas que no sea óptimo (esto es, el conjunto contiene más monedas que las necesarias), mientras que quizá en otros casos no llegue a encontrar ninguna solución aun cuando ésta exista (aunque esto no puede suceder si disponemos de un suministro ilimitado de monedas de una unidad).

El algoritmo es «voraz» porque en cada paso selecciona la mayor de las monedas que pueda encontrar, sin preocuparse por lo correcto de esta decisión a la larga. Además, nunca cambia de opinión: una vez que una moneda se ha incluido en la solución, la tal moneda queda allí para siempre. Tal como explicaremos en la sección siguiente, éstas son las características de esta familia de algoritmos.

Para el problema particular de devolver cambio, se describe un algoritmo completamente distinto en el Capítulo 8. Este algoritmo alternativo emplea programación dinámica. El algoritmo de programación dinámica funciona siempre, mientras que el algoritmo voraz puede fallar; sin embargo, es menos comprensible que el algoritmo voraz y (cuando funcionan ambos algoritmos) resulta menos eficiente.

6.2 CARACTERÍSTICAS GENERALES DE LOS ALGORITMOS VORACES

Generalmente, los algoritmos voraces y los problemas que éstos resuelven *se caracterizan por la mayoría de las propiedades* siguientes (o quizá por todas):

Tenemos que resolver algún problema de forma óptima. Para construir la solución de nuestro problema, disponemos de un conjunto (o una lista) de candidatos: las monedas disponibles, las aristas de un grafo que se pueden utilizar para construir una ruta, el conjunto de tareas que hay que planificar, o lo que sea.

A medida que avanza el algoritmo, vamos acumulando dos conjuntos. Uno contiene candidatos que ya han sido considerados y seleccionados, mientras que el otro contiene candidatos que han sido considerados y rechazados.

Existe una función que comprueba si un cierto conjunto de candidatos constituye una *solución* de nuestro problema, ignorando si es o no óptima por el momento. Por ejemplo, ¿suman las monedas seleccionadas la cantidad que hay que pagar? ¿Proporcionan las aristas seleccionadas una ruta hasta el nodo que deseamos alcanzar? ¿Están planificadas todas las tareas?

Hay una segunda función que comprueba si un cierto conjunto de candidatos es *factible*, esto es, si es posible o no completar el conjunto añadiendo otros candidatos para obtener al menos una solución de nuestro problema. Una vez más, no nos preocupa aquí si esto es óptimo o no. Normalmente, se espera que el problema tenga al menos una solución que sea posible obtener empleando candidatos del conjunto que estaba disponible inicialmente.

Hay otra función más, la *función de selección*, que indica en cualquier momento cuál es el más prometedor de los candidatos restantes, que no han sido seleccionados ni rechazados.

Por último, existe una *función objetivo* que da el valor de la solución que hemos hallado: el número de monedas utilizadas para dar la vuelta, la longitud de la ruta que hemos construido, el tiempo necesario para procesar todas las tareas de la planificación, o cualquier otro valor que estemos intentando optimizar. A diferencia de las tres funciones mencionadas anteriormente, la función objetiva no aparece explícitamente en el algoritmo voraz.

Para resolver nuestro problema, buscamos un conjunto de candidatos que constituya una solución, y que optimice (minimice o maximice, según los casos) el valor de la *función objetivo*. Los algoritmos voraces avanzan paso a paso. Inicialmente, el conjunto de elementos seleccionados está vacío. Entonces, en cada paso se considera añadir a este conjunto el mejor candidato sin *considerar los restantes*, estando guiada nuestra elección por la función de selección. Si el conjunto ampliado de candidatos seleccionados ya no fuera factible, rechazamos el candidato que estamos considerando en ese momento. Sin embargo, si el conjunto aumentado sigue siendo factible, entonces añadimos el candidato actual al conjunto de candidatos selecciona-

dos, en donde pasará a estar desde ahora en adelante. Cada vez que se amplía el conjunto de candidatos seleccionados, comprobamos si éste constituye ahora una solución para nuestro problema. Cuando el algoritmo voraz funciona correctamente, la primera solución que se encuentre de esta manera es siempre óptima:

```
función voraz(C: conjunto): conjunto
    {C es el conjunto de candidatos}
    S ← ∅ {Construimos la solución en el conjunto S}
    mientras C ≠ ∅ y no solución(s) hacer
        x ← seleccionar(C)
        C ← C \ {x}
        si factible(S ∪ {x}) entonces S ← S ∪ {x}
    si solución(S) entonces devolver S
                   sino devolver «no hay soluciones»
```

Está clara la razón por la cual tales algoritmos se denominan «voraces»: en cada paso, el procedimiento selecciona el mejor bocado que pueda tragar, sin preocuparse por el futuro. Nunca cambia de opinión: una vez que un candidato se ha incluido en la solución, queda allí para siempre; una vez que se excluye un candidato de la solución, nunca vuelve a ser considerado.

La función de selección suele estar relacionada con la función objetivo. Por ejemplo, si estamos intentando maximizar nuestros beneficios, es probable que seleccionemos aquel candidato restante que posea un mayor valor individual. Si intentamos minimizar el coste, entonces quizá seleccionemos el más barato de los candidatos disponibles, y así sucesivamente. Sin embargo, veremos que en algunas ocasiones puede haber varias funciones de selección plausibles, así que hay que seleccionar la adecuada si deseamos que nuestro algoritmo funcione correctamente.

Volviendo por un momento al ejemplo de dar cambio, lo que sigue es una forma de adecuar las características generales de los algoritmos voraces a las características particulares de este problema.

- ◊ Los candidatos son un conjunto de monedas, que representan en nuestro ejemplo 100, 25, 10, 5 y 1 pesetas, con tantas monedas de cada valor que nunca las agotamos. (Sin embargo, el conjunto de candidatos debe ser finito.)
- ◊ La función de solución comprueba si el valor de las monedas seleccionadas hasta el momento es *exactamente* el valor que hay que pagar.
- ◊ Un conjunto de monedas será *factible* si su valor total *no sobrepasa* la cantidad que haya que pagar.
- ◊ La función de selección toma la moneda de valor más alto que quede en el conjunto de candidatos.
- ◊ La función objetivo cuenta el número de monedas utilizadas en la solución.

Está claro que es más eficiente rechazar todas las monedas restantes de 100 pesetas (digamos) cuando el valor restante que hay que pagar cae por debajo de ese valor. El uso de la división entera para calcular cuántas monedas de un cierto valor hay que tomar también es más eficiente que actuar por sustracciones sucesivas. Si se adopta cualquiera de estas tácticas, entonces podemos relajar la condición consistente en que el conjunto de monedas debe ser finito.

6.3 GRAFOS: ÁRBOLES DE RECUBRIMIENTO MÍNIMO

Sea $G = \langle N, A \rangle$ un grafo conexo no dirigido en donde N es el conjunto de nodos y A es el conjunto de aristas. Cada arista posee una *longitud* no negativa. El problema consiste en hallar un subconjunto T de las aristas de G tal que utilizando solamente las aristas de T, todos los nodos deben quedar conectados, y además la suma de las longitudes de las aristas de T debe ser tan pequeña como sea posible. Dado que G es conexo, debe existir al menos una solución. Si G tiene aristas de longitud 0, pueden existir varias soluciones cuya longitud total sea la misma, pero que tengan números distintos de aristas. En este caso, dadas dos soluciones de igual longitud total, preferimos la que contenga menos aristas. Incluso con esta condición el problema puede tener varias soluciones diferentes y de igual valor. En lugar de hablar de las longitudes, podemos asociar un *coste* a cada arista. El problema, entonces, consiste en hallar un subconjunto T de las aristas cuyo coste total sea el menor posible. Evidentemente, este cambio de terminología no afecta a la forma en que resolvemos el problema.

Sea $G' = \langle N, T \rangle$ el grafo parcial formado por los nodos de G y las aristas de T, y supongamos que en N hay n nodos. Un grafo conexo con n nodos debe tener al menos $n - 1$ aristas, así que éste es el número mínimo de aristas que puede haber en T. Por otra parte, un grafo con n nodos y más de $n - 1$ aristas contiene al menos un ciclo; véase el Problema 6.7. Por tanto, si G' es conexo y T tiene más de $n - 1$ aristas, se puede eliminar al menos una arista sin desconectar G', siempre y cuando seleccionemos una arista que forme parte de un ciclo. Esto hará o bien que disminuya la longitud total de las aristas de T, o bien que la longitud total quede intacta (si hemos eliminado una arista de longitud 0) a la vez que disminuye el número de aristas que hay en T. En ambos casos, la nueva solución es preferible a la anterior. Por tanto, un conjunto T con n o más aristas no puede ser óptimo. Se sigue que T debe tener exactamente $n - 1$ aristas, y como G' es conexo, tiene que ser un árbol.

El grafo G' se denomina *árbol de recubrimiento*[2] *mínimo* para el grafo G. Este problema tiene muchas aplicaciones. Por ejemplo, supongamos que los nodos de G representan ciudades, y sea el coste de una arista $\{a, b\}$ el coste de tender una línea telefónica desde a hasta b. Entonces un árbol de recubrimiento mínimo de G se corresponde con la red más barata posible para dar servicio a todas las ciudades en cuestión, siempre y cuando sólo se puedan utilizar conexiones directas entre ciudades (en otras palabras, siempre y cuando no se permita establecer centrales te-

lefónicas en el campo, *entre* las ciudades). Relajar esta condición es equivalente a permitir que se añadan nodos adicionales, auxiliares, a G. Esto puede permitirnos obtener soluciones más baratas: véase el problema 6.8.

A primera vista, parece que existen al menos dos líneas de ataque si deseamos encontrar un algoritmo voraz para este problema. Está claro que nuestro conjunto de candidatos debe ser el conjunto A de aristas que están en G. Una táctica posible consiste en comenzar por un conjunto vacío T, y seleccionar en cada etapa la arista más corta que todavía no haya sido seleccionada o rechazada, independientemente de donde se encuentra esta arista en G. Otra línea de ataque implica seleccionar un nodo y construir un árbol a partir de él, seleccionando en cada etapa la arista más corta posible que pueda extender el árbol hasta un nodo adicional. Es poco frecuente, pero ¡para este problema concreto, funcionan las dos aproximaciones! Antes de presentar los algoritmos, mostraremos la forma en que se aplica a este caso el esquema general de los algoritmos voraces, y presentaremos un lema que será de utilidad posteriormente:

◊ Los candidatos, como ya se ha indicado, son las aristas de G.
◊ Un conjunto de aristas es una solución si constituye un árbol de recubrimiento para los nodos de N.
◊ Un conjunto de aristas es factible si no contiene ningún ciclo.
◊ La función de selección que utilizamos varía con el algoritmo.
◊ La función objetivo que hay que minimizar es la longitud total de las aristas de la solución.

También necesitamos algo más de terminología. Diremos que un conjunto de aristas factible es *prometedor* si se puede extender para producir no sólo una solución, sino una solución óptima para nuestro problema. En particular, el conjunto vacío siempre es prometedor (puesto que siempre existe una solución óptima). Además, si un conjunto de aristas prometedor ya es una solución, la extensión requerida es irrelevante, y esta solución debe ser óptima . Decimos que una arista *sale* de un conjunto de nodos dado si esa arista tiene exactamente un extremo en el conjunto de nodos. Por tanto, una arista puede no salir de un conjunto dado de nodos bien porque ninguno de sus extremos esté en el conjunto, bien porque —y esto es menos evidente— sus dos extremos se encuentren en el conjunto. El lema siguiente es crucial para demostrar la corrección de los próximos algoritmos.

Lema 6.3.1. *Sea* $G=\langle N, A\rangle$ *un grafo conexo no dirigido en el cual está dada la longitud de todas las aristas. Sea* $B \subset N$ *un subconjunto estricto de los nodos de* G. *Sea* $T \subseteq A$ *un conjunto prometedor de aristas tal que no haya ninguna arista de* T *que sale de* B. *Sea* v *la arista más corta que salga de* B *(o una de las más cortas si hay empates). Entonces* $T \cup \{v\}$ *es prometedor.*

Demostración. Sea U un árbol de recubrimiento mínimo de G tal que $T \subseteq U$. Este U tiene que existir puesto que T es prometedor por hipótesis. Si $v \in U$, entonces no hay nada que probar. Si no, cuando añadimos v a U creamos exactamente un ciclo. (Ésta es una propiedad de los árboles: véase la Sección 5.5.) En este ciclo, puesto que v sale de B, existe necesariamente al menos otra arista u que también sale de B, o bien el ciclo no se cerraría; véase la figura 6.1. Si ahora eliminamos u, el ciclo desaparece y obtenemos un nuevo árbol V que abarca G. Sin embargo, la longitud de v, por definición, no es mayor que la longitud de u, y por tanto la longitud total de las aristas de V no sobrepasa la longitud total de las aristas de U. Por tanto, V es también un árbol de recubrimiento mínimo de G y contiene a v. Para completar la demostración, sólo hay que comentar que $T \subseteq V$ porque la arista u que se ha eliminado sale de B, y por tanto no podría haber sido una arista de T.

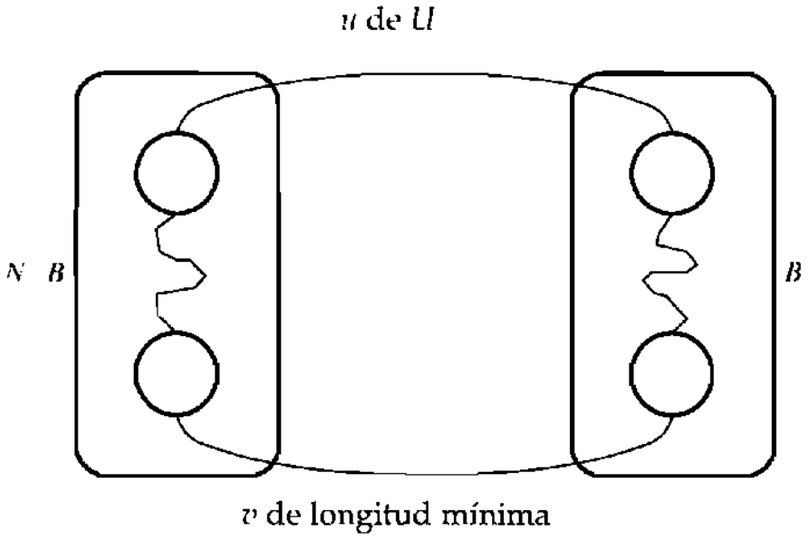


Figura 6.1. Se crea un ciclo si añadimos una arista v a U

6.3.1 Algoritmo de Kruskal

El conjunto de aristas T está vacío inicialmente. A medida que progresa el algoritmo, se van añadiendo aristas a T. Mientras no haya encontrado una solución, el grafo parcial formado por los nodos de G y las aristas de T consta de varios componentes conexos. (Inicialmente, cuando T está vacío, cada nodo de G forma una componente conexa distinta trivial.) Los elementos de T que se incluyen en una componente conexa dada forman un árbol de recubrimiento mínimo para los nodos de esta componente. Al final del algoritmo, sólo queda una componente conexa, así que T es un árbol de recubrimiento mínimo para todos los nodos de G.

Para construir componentes conexas más y más grandes, examinamos las aristas de G por orden creciente de longitudes. Si una arista une a dos nodos de compo-

nentes conexas distintas, se lo añadimos a T. Consiguientemente, las dos componentes conexas forman ahora una única componente. En caso contrario, se rechaza la arista: une a dos nodos de la misma componente conexa, y por tanto no se puede añadir a T sin formar un ciclo (porque las aristas de T forman un árbol para cada componente). El algoritmo se detiene cuando sólo queda una componente conexa.

Para ilustrar la forma en que funciona este algoritmo, considérese el gráfico de la figura 6.2. Por orden creciente de longitud, las aristas son: {1,2}, {2,3}, {4,5}, {6,7}, {1,4}, {2,5}, {4,7}, {3,5}, {2,4}, {3,6}, {5,7} y {5,6}.

Paso	Arista considerada	Componentes conexas
Iniciación	—	{1} {2} {3} {4} {5} {6} {7}
1	{1, 2}	{1, 2} {3} {4} {5} {6} {7}
2	{2, 3}	{1, 2, 3} {4} {5} {6} {7}
3	{4, 5}	{1, 2, 3} {4, 5} {6} {7}
4	{6, 7}	{1, 2, 3} {4, 5} {6, 7}
5	{1, 4}	{1, 2, 3, 4, 5} {6, 7}
6	{2,5}	rechazado
7	{4, 7}	{1, 2, 3, 4, 5, 6, 7}

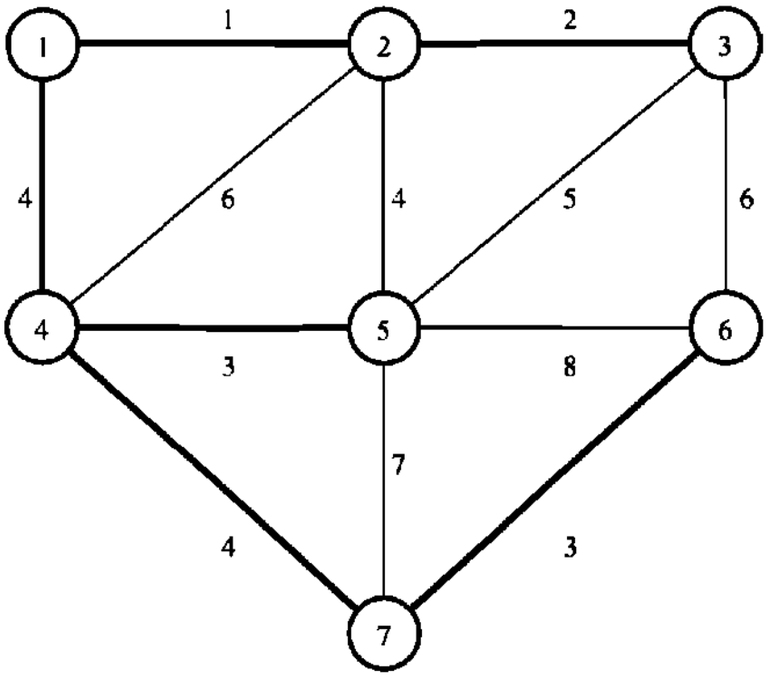


Figura 6.2 Un grafo y su árbol de recubrimiento mínimo

Cuando se detiene el algoritmo, T contiene las aristas seleccionadas {1,2}, {2,3}, {4,5}, {6,7}, {1,4} y {4,7}. El árbol de recubrimiento mínimo se muestra mediante trazos gruesos en la figura 6.2; la longitud total es 17.

Teorema 6.3.2 *El algoritmo de Kruskal halla un árbol de recubrimiento mínimo*

Demostración. La demostración se hace por inducción matemática sobre el número de aristas que hay en el conjunto T. Mostraremos que si T es prometedor entonces sigue siendo prometedor en cualquier fase del algoritmo cuando se le añade una arista adicional. Cuando se detiene el algoritmo, T da una solución de nuestro problema; puesto que también es prometedora, la solución es óptima.

◊ *Base*: el conjunto vacío es prometedor porque G es conexo, y por tanto tiene que existir una solución.

◊ *Paso de inducción*: supongamos que T es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $e = \{u, v\}$. Las aristas de T dividen a los nodos de G en dos o más componentes conexas; el nodo u se encuentra en una de estas componentes, y v está en otra componente distinta. Sea B el conjunto de nodos de esa componente que contiene a u. Ahora:
- el conjunto B es un subconjunto estricto de los nodos de G (puesto que no incluye a v, por ejemplo);
- T es un conjunto prometedor de aristas tal que ninguna arista de T sale de B (porque una arista de T tiene o bien ambas aristas en B, o bien ningúna arista en B, así que por definición no sale de B); y
- e es una de las aristas más cortas que salen de B (porque todas las aristas estrictamente más cortas ya se han examinado, y o bien se han añadido a T, o bien se han rechazado porque tenían los dos extremos en la misma componente conexa.

Entonces se cumplen las condiciones del Lema 6.3.1, y concluimos que el conjunto $T \cup \{e\}$ también es prometedor.

Esto completa la demostración por inducción matemática de que el conjunto T es prometedor en todas las fases del algoritmo, y por tanto cuando se detiene el algoritmo, T no da meramente una solución de nuestro problema, sino una solución óptima.

Para implementar el algoritmo, tenemos que manejar un cierto número de conjuntos, a saber, los nodos de cada componente conexa. Es preciso efectuar rápidamente dos operaciones: *buscar*(x), que nos dice en qué componente conexa se encuentra el nodo x, y *fusionar*(A, B), para fusionar dos componentes conexas. Por tanto, utilizamos estructuras de partición; véase la Sección 5.9. Para este algoritmo, es preferible representar el grafo como un vector de aristas con sus longitudes asociadas, y no como una matriz de distancias; véase el problema 6.9. El algoritmo es el siguiente:

```
función Kruskal(G = ⟨N, A⟩: grafo; longitud: A → ℝ⁺): conjunto de aristas
    {Iniciación}
    Ordenar A por longitudes crecientes
    n ← el número de nodos que hay en N
    T ← Ø {contendrá las aristas del árbol de recubrimiento mínimo}
    Iniciar n conjuntos, cada uno de los cuales contiene un elemento distinto de N
    {bucle voraz}
    repetir
        e ← {u, v} ← arista más corta, aun no considerada
        compu ← buscar(u)
        compv ← buscar(v)
        si compu ≠ compv entonces
            fusionar(compu, compv)
            T ← T ∪ {e}
    hasta que T contenga n - 1 aristas
    devolver T
```

Podemos evaluar el tiempo de ejecución del algoritmo en la forma siguiente. En un grafo con n nodos y a aristas, el número de operaciones está en:

- ◊ $\Theta(a \log a)$ para ordenar las aristas, lo cual es equivalente a $\Theta(a \log n)$ porque $n - 1 \leq a \leq n(n-1)/2$
- ◊ $\Theta(n)$ para iniciar los n conjuntos disjuntos
- ◊ $\Theta(2a\ \alpha\ (2a, n))$ para todas las operaciones *buscar* y *fusionar*, en donde α es la función de crecimiento lento de la Sección 5.9 (esto se sigue de los resultados de la Sección 5.9, puesto que hay como máximo $2a$ operaciones *buscar* y $n-1$ operaciones *fusionar* en un universo de n elementos), y
- ◊ $O(a)$ en el caso peor, para las operaciones restantes.

Concluimos que el tiempo total para el algoritmo está en $\Theta(a \log n)$ porque $\Theta(\alpha(2a, n)) \subset \Theta(\log n)$. Aunque esto no cambia el análisis en el caso peor, es preferible mantener las aristas en un montículo invertido (véase la Sección 7.5): de esta manera la arista más corta se encuentra en la raíz del montículo. Esto permite efectuar la inicialización en un tiempo que está en $\Theta(a)$, aun cuando cada búsqueda de un mínimo en el bucle **repetir** requiere ahora un tiempo que está en $\Theta(\log a) = \Theta(\log n)$. Esto resulta especialmente ventajoso si se encuentra el árbol de recubrimiento mínimo en un momento en el que quede por probar un número considerable de aristas. En tales casos, el algoritmo original desperdicia el tiempo ordenando estas aristas inútiles.

6.3.2 Algoritmo de Prim

En el algoritmo de Kruskal, la función de selección escoge las aristas por orden creciente de longitud, sin preocuparse demasiado por su conexión con las aristas seleccionadas anteriormente, salvo que se tiene cuidado para no formar nunca un ciclo. El resultado es un bosque de árboles que crece al azar, hasta que finalmente todas las componentes del bosque se fusionan en un único árbol. En el algoritmo de Prim, por

otra parte, el árbol de recubrimiento mínimo crece de forma natural, comenzando por una raíz arbitraria. En cada fase se añade una nueva rama al árbol ya construido; el algoritmo se detiene cuando se han alcanzado todos los nodos.

Sea B un conjunto de nodos, y sea T un conjunto de aristas. Inicialmente, B contiene un único nodo arbitrario, y T está vacío. En cada paso, el algoritmo de Prim busca la arista más corta posible $\{u, v\}$ tal que $u \in B$ y $v \in N \backslash B$. Entonces añade v a B y $\{u, v\}$ a T. De esta manera las aristas de T forman en todo momento un árbol de recubrimiento mínimo para los nodos de B. Continuamos mientras $B \neq N$. Lo que sigue es un enunciado informal del algoritmo:

función *Prim*($G = \langle N, A \rangle$: *grafo*; *longitud* $A \rightarrow \mathbb{R}^+$): *conjunto de aristas*.
  {Iniciación}
  $T \leftarrow \emptyset$
  $B \leftarrow$ {un miembro arbitrario de N}
  mientras $B \neq N$ **hacer**
    buscar $e = \{u, v\}$ de *longitud* mínima tal que
      $u \in B$ y $v \in N \setminus B$
    $T \leftarrow T \cup \{e\}$
    $B \leftarrow B \cup \{v\}$
  devolver T

Para ilustrar este algoritmo, considérese una vez más el grafo de la figura 6.2. Arbitrariamente, seleccionamos el nodo uno como nodo inicial. Ahora el algoritmo podría progresar en la forma siguiente:

Paso	$\{u, v\}$	B
Inicialización	—	{1}
1	{1, 2}	{1, 2}
2	{2, 3}	{1, 2, 3}
3	{1, 4}	{1, 2, 3, 4}
4	{4, 5}	{1, 2, 3, 4, 5}
5	{4, 7}	{1, 2, 3, 4, 5, 7}
6	{7, 6}	{1, 2, 3, 4, 5, 6, 7}

Cuando se detiene el algoritmo, T contiene las aristas seleccionadas {1, 2}, {2, 3}, {1,4}, {4, 5}, {4, 7} y {7, 6}. La demostración de que el algoritmo funciona es parecida a la demostración del algoritmo de Kruskal.

Teorema 6.3.3 *El algoritmo de Prim halla un árbol de recubrimiento mínimo*

Demostración. La demostración es por inducción matemática sobre el número de aristas que hay en el conjunto T. Demostraremos que si T es prometedor en alguna fase del algoritmo,

entonces sigue siendo prometedor al añadir una arista adicional. Cuando se detiene el algoritmo, T da una solución para nuestro problema; puesto que también es prometedora, la solución es óptima.

◊ *Base*: el conjunto vacío es prometedor.
◊ *Paso de inducción*: suponga que T es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $e = \{u, v\}$. Ahora B es un subconjunto estricto de N (porque el algoritmo se detiene cuando $B = N$), T es un conjunto de aristas prometedor por hipótesis de inducción, y e es por definición una de las aristas más cortas que salen de B. Entonces las condiciones del lema 6.3.1 se cumplen, y $T \cup \{e\}$ también es prometedor.

Esto completa la demostración por inducción matemática de que el conjunto T es prometedor en todas las fases del algoritmo. Por tanto, cuando se detiene el algoritmo, T ofrece una solución óptima de nuestro problema.

Para obtener una implementación sencilla en una computadora, supongamos que los nodos de G están numerados de 1 a n, así que $N = \{1, 2, \ldots, n\}$. Supongamos también que hay una matriz simétrica L que da la longitud de todas las aristas, con $L[i, j] = \infty$ si no existe la arista correspondiente. Utilizaremos dos matrices. Para todo nodo $i \in N \setminus B$, *más próximo*$[i]$ proporciona el nodo de B que está más próximo a i, y *distmin*$[i]$ da la distancia desde i hasta este nodo más próximo. Para un nodo $i \in B$, hacemos *distmin*$[i] = -1$. (De esta manera, podemos saber si un nodo está o no en B.) El conjunto B, que recibe el valor inicial arbitrario $\{1\}$, no se representa explícitamente; *más próximo*$[1]$ y *distmin*$[1]$ nunca se utilizan. Véase el algoritmo:

```
función Prim(L[1..n, 1..n]): conjunto de aristas
    {Inicialización: sólo el nodo 1 se encuentra en B}
    T ← ∅ {contendrá las aristas del árbol de recubrimiento mínimo}
    para i ← 2 hasta n hacer
        más próximo[i] ← 1
        distmin[i] ← L[i, 1]
    {bucle voraz}
    repetir n – 1 veces
        mín ← ∞
        para j ← 2 hasta n hacer
            si 0 ≤ distmin[j] < mín entonces mín ← distmin[j]
                                             k ← j
        T ← T ∪ {{más próximo[k], k}}
        distmin[k] ← –1 {se añade k a B}
        para j ← 2 hasta n hacer
            si L[j, k] < distmin[j] entonces distmin[k] ← L[j, k]
                                             más próximo[j] ← k
        devolver T
```

El bucle principal del algoritmo se ejecuta n - 1 veces; en cada iteración, el bucle **para** anidado requiere un tiempo que está en $\Theta(n)$. Por tanto, el algoritmo de Prim requiere un tiempo que está en $\Theta(n^2)$.

Se vio anteriormente que el algoritmo de Kruskal requiere un tiempo que está en $\Theta(a \log n)$, en donde a es el número de aristas que hay en el grafo. Para un grafo denso, a tiende a $n(n-1)/2$. En este caso, el algoritmo de Kruskal requiere un tiempo que se encuentra en $\Theta(n^2 \log n)$, y el algoritmo de Prim puede ser mejor. Para un grafo disperso, a tiende a n. En este caso, el algoritmo de Kruskal requiere un tiempo que está en $\Theta(n \log n)$, y el algoritmo de Prim, tal como se ha presentado aquí, es probablemente menos eficiente. Sin embargo, el algoritmo de Prim, al igual que el de Kruskal, se puede implementar utilizando montículos. En este caso —una vez más, como el algoritmo de Kruskal— requiere un tiempo que está en $\Theta(a \log n)$. Existen otros algoritmos más eficientes que el de Prim o el de Kruskal; véase la Sección 6.8.

6.4 GRAFOS: CAMINOS MÍNIMOS

Considere ahora un grafo dirigido $G = \langle N, A \rangle$ en donde N es el conjunto de nodos de G, y A es el conjunto de aristas dirigidas. Cada arista posee una *longitud* no negativa. Se toma uno de los nodos como nodo *origen*. El problema consiste en determinar la longitud del camino mínimo que va desde el origen hasta cada uno de todos los demás nodos del grafo. Tal como sucedía en la Sección 6.3, podríamos hablar igualmente bien acerca del *coste* de una arista, en lugar de mencionar su longitud, y se podría plantear el problema consistente en determinar la ruta más barata desde el origen hasta cada uno de todos los demás nodos.

Este problema se puede resolver mediante un algoritmo voraz que recibe frecuentemente el nombre de *algoritmo de Dijkstra*. El algoritmo utiliza dos conjuntos de nodos, S y C. En todo momento, el conjunto S contiene aquellos nodos que ya han sido seleccionados; como veremos, la distancia mínima desde el origen ya es conocida para todos los nodos de S. El conjunto C contiene todos los demás nodos, cuya distancia mínima desde el origen todavía no es conocida, y que son candidatos a ser seleccionados en alguna etapa posterior. Por tanto, tenemos la propiedad invariante $N = S \cup C$. En un primer momento, S contiene nada más el origen en sí; cuando se detiene el algoritmo, S contiene todos los nodos del grafo y nuestro problema está resuelto. En cada paso seleccionamos aquel nodo de C cuya distancia al origen sea mínima, y se lo añadimos a S.

Diremos que un camino desde el origen hasta algún otro nodo es *especial* si todos los nodos intermedios a lo largo del camino pertenecen a S. En cada fase del algoritmo, hay una matriz D que contiene la longitud del camino especial más corta que va hasta cada nodo del grafo. En el momento en que se desea añadir un nuevo nodo v a S, el camino especial mas corto hasta v es también el más corto de los caminos posibles hasta v (esto se demostrará más adelante). Cuando se detiene el algoritmo, todos los nodos del grafo se encuentran en S, y por tanto todos los caminos desde el origen hasta algún otro nodo son especiales. Consiguientemente, los valores que hay en D dan la solución del problema de caminos mínimos.

Por sencillez, suponemos una vez más que los nodos de G están numerados desde 1 hasta n, así que $N = \{1, 2,..., n\}$. Podemos suponer sin pérdida de generalidad que el nodo uno es el nodo origen. Supongamos también que la matriz L da la longitud de todas las aristas dirigidas: $L[i, j] \geq 0$ si la arista $(i, j) \in A$, y $L[i, j] = \infty$ en caso contrario. Véase a continuación el algoritmo:

```
función Dijkstra(L[1..n, 1..n]): matriz [2..n]
   matriz D[2..n]
   {Iniciación}
   C ← {2, 3,..., n} {S = N \ C sólo existe implícitamente}
   para i ← 2 hasta n hacer D[i] ← L[1, i]
   {bucle voraz}
   repetir n - 2 veces
      v ← algún elemento de C que minimiza D[v]
      C ← C \ {v}  {e implícitamente S ← S ∪ {v}}
      para cada w ∈ C hacer
         D[w] ← mín(D[w], D[v] + L[v, w])
   devolver D
```

El algoritmo procede en la forma siguiente para el grafo de la figura 6.3.

Paso	v	C	D
Inicialización	—	{2, 3, 4, 5}	[50, 30, 100, 10]
1	5	{2, 3, 4}	[50, 30, 20, 10]
2	4	{2, 3}	[40, 30, 20, 10]
3	3	{2}	[35, 30, 20, 10]

Claramente, D no cambiaría si hiciéramos una iteración más para eliminar el último elemento de C. Ésta es la razón por la cual el bucle principal se repite solamente $n - 2$ veces.

Para determinar no solamente la longitud de los caminos mínimos sino también por dónde pasan, se añade una segunda matriz $P[2..n]$, en donde $P[v]$ contiene el número del nodo que precede a v dentro del camino más corto. Para hallar el camino más corto, se siguen los punteros P hacia atrás, desde el destino hacia el origen. Las modificaciones necesarias para el algoritmo son sencillas:

Se inicia $P[i]$ con el valor 1 para $i = 2, 3,..., n$
Se sustituye el contenido del bucle interno **para** por

```
si D[w] > D[v] + L[v, w] entonces  D[w] ← D[v] + L[v, w]
                                   P[w] ← v
```

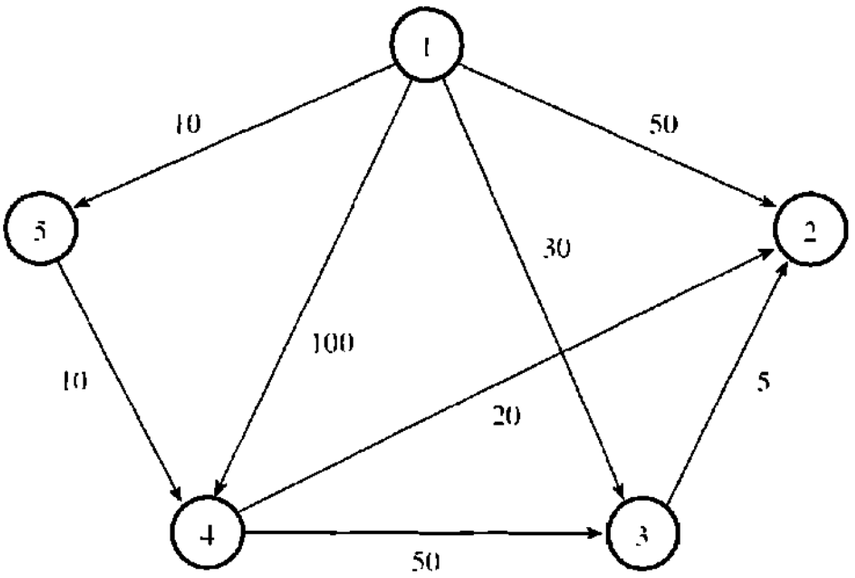


Figura 6.3. Un grafo dirigido

La demostración de que el algoritmo funciona es, una vez más, por inducción matemática.

Teorema 6.4.1 *El algoritmo de Dijkstra halla los caminos más cortos desde un único origen hasta los demás nodos del grafo*

Demostración. Demostraremos por inducción matemática que:

(a) Si un nodo $i \neq 1$ está en S, entonces $D[i]$ da la longitud del camino más corto desde el origen hasta i, y

(*b*) si un nodo i no está en S, entonces $D[i]$ da la longitud del camino *especial* más corto desde el origen hasta i.

◊ *Base*: inicialmente, sólo el nodo 1, que es el origen, se encuentra en S, así que la situación (a) es cierta sin más demostración. Para los demás nodos, el único camino especial desde el origen es el camino directo, y D recibe valores iniciales en consecuencia. Por tanto la situación (b) también es cierta cuando comienza el algoritmo.

◊ *Hipótesis de inducción*: la hipótesis de inducción es que tanto la situación (a) como la situación (b) son válidas inmediatamente antes de añadir un nodo v a S. Detallamos por separado los pasos de inducción para las situaciones (a) y (b).

◊ *Paso de inducción para la situación (a)*: para todo nodo que ya esté en S antes de añadir v, no cambia nada, así que la situación (a) sigue siendo válida. En cuanto al nodo v, ahora pertenecerá a S. Antes de añadirlo a S, es preciso comprobar que $D[v]$ proporcione la longitud del camino más corto que va desde el origen hasta v. Por hipótesis de inducción, nos da ciertamente la longitud del camino *especial* más corto. Por tanto, hay que verificar que el camino más corto desde el origen hasta v no pase por ninguno de los nodos que no pertenecen a S.

Supongamos lo contrario; esto es, supongamos que cuando se sigue el camino más corto desde el origen hasta v, se encuentran uno o más nodos (sin contar el propio v)

que no pertenecen a S. Sea x el primer nodo encontrado con estas características; véase la figura 6.4. Ahora el segmento inicial de esta ruta, hasta llegar a x, es una ruta especial, así que la distancia hasta x es $D[x]$ por la parte (b) de la hipótesis de inducción. Claramente la distancia total hasta v a través de x no es más corta que este valor, porque las longitudes de las aristas son no negativas. Finalmente, $D[x]$ no es menor que $D[v]$, porque el algoritmo ha seleccionado a v antes que a x. Por tanto, la distancia total hasta v a través de x es como mínimo $D[v]$, y el camino a través de x no puede ser más corto que el camino especial que lleva hasta v.

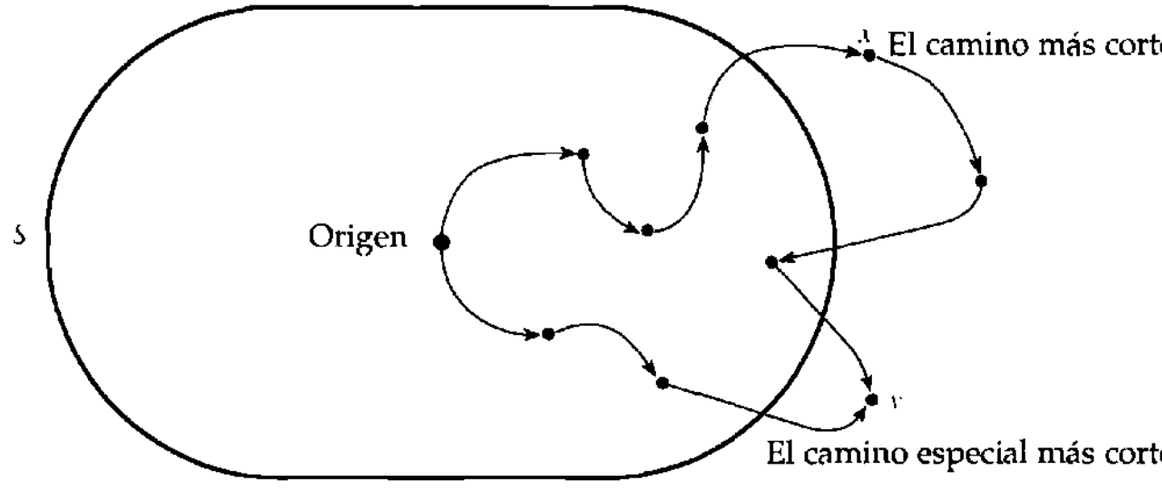


Figura 6.4. El camino más corto hasta v no puede visitar x

Por tanto, hemos verificado que cuando se añade v a S, la parte (a) de la inducción sigue siendo válida.

◊ *Paso de inducción para la situación* (b): considérese ahora un nodo w, distinto de v, que no se encuentre en S. Cuando v se añade a S, hay dos posibilidades para el camino especial más corto desde el origen hasta w: o bien no cambia, o bien ahora pasa a través de v (y posiblemente, también a través de otros nodos de S). En el segundo caso, sea x el último nodo de S visitado antes de llegar a w. La longitud de este camino es $D[x] + L[x, w]$. Parece a primera vista que para calcular el nuevo valor de $D[w]$ debiéramos comparar el valor anterior de $D[w]$ con los valores de $D[x] + L[x, w]$ para todo nodo x de S (incluyendo a v). Sin embargo, para todos los nodos x de S salvo v, esta comparación se ha hecho cuando se añadió x a S, y $D[x]$ no ha variado desde entonces. Por tanto, el nuevo valor de $D[w]$ se puede calcular sencillamente comparando el valor anterior con $D[v] + L[v, w]$.

Puesto que el algoritmo hace esto explícitamente, asegura que la parte (b) de la inducción siga siendo cierta también cuando se añade a S un nuevo nodo v.

Para completar la demostración de que el algoritmo funciona, obsérvese que cuando se detenga el algoritmo, todos los nodos menos uno estarán en S (aun cuando el conjunto S no se construye explícitamente). En ese momento queda claro que el camino más corto desde el origen hasta el nodo restante es un camino especial.

Análisis del algoritmo. Supongamos que el algoritmo de Dijkstra se aplica a un grafo que posee n nodos y a aristas. Utilizando la representación sugerida hasta el momento, este caso se da en la forma de una matriz $L[1..n, 1..n]$. La iniciación requiere un tiempo que está en $O(n)$. En una implementación directa, la selección de v dentro del bucle **(repetir)** requiere examinar todos los elementos de C, así que examinaremos $n-1, n-2,..., 2$ valores de D en las sucesivas iteraciones, dando un tiempo total que está en $\Theta(n^2)$. El bucle **para (desde)** interno realiza $n-2, n-3,...,$ 1 iteraciones dando también un tiempo total que está en $\Theta(n^2)$. El tiempo requerido por esta versión del algoritmo está, por tanto, en $\Theta(n^2)$.

Si $a << n^2$, podríamos tener la esperanza de evitar examinar las muchas entradas que contienen ∞ en la matriz L. Teniendo esto en cuenta, sería preferible representar el grafo mediante una matriz de n listas, que diera para cada nodo su distancia directa a los nodos adyacentes (como el tipo *listagraf* de la Sección 5.4). Esto nos permite ahorrar tiempo en el bucle **para** interno, porque sólo hay que considerar aquellos nodos w que sean adyacentes a v; ahora bien, ¿cómo podemos evitar que se necesite un tiempo en $\Omega(n^2)$ para determinar sucesivamente los $n-2$ valores que va tomando v?

La respuesta consiste en utilizar un montículo invertido que contiene un nodo para cada elemento v de C, ordenado por el valor de $D[v]$. De esta manera, el elemento v de C que minimiza $D[v]$ siempre se encontrará en la raíz. La iniciación del montículo requiere un tiempo que está en $\Theta(n)$. La instrucción «$C \leftarrow C \setminus \{v\}$» consiste en eliminar la raíz del montículo, lo cual requiere un tiempo que se encuentra en $O(\log n)$. En cuanto al bucle **para** interno, ahora consiste en estudiar, para cada elemento w de C adyacente a v, si $D[v] + L[v, w]$ es menor que $D[w]$. En tal caso, debemos modificar $D[w]$ y filtrar a w dentro del montículo, lo cual requiere una vez más un tiempo que está en $O(\log n)$. Esto no sucede más que una vez para cada arista del grafo.

Para resumir, tenemos que eliminar la raíz del montículo exactamente $n-2$ veces, y tenemos que flotar un máximo de a nodos, lo cual da un tiempo total que está en $\Theta((a+n) \log n)$. Si el grafo es conexo, $a \geq n-1$, y el tiempo se encuentra en $\Theta(a \log n)$. La implementación sencilla es preferible, por tanto, si el grafo es denso, mientras que resulta preferible utilizar un montículo si el grafo es disperso. Si $a \in \Theta(n^2/\log n)$, entonces la elección de la representación puede depender de la implementación concreta utilizada. El Problema 6.16 sugiere una forma de acelerar el algoritmo, empleando un montículo k-ario con un valor de k bien seleccionado; se conocen otros algoritmos aún más rápidos; véanse el Problema 6.17 y la Sección 6.8.

6.5 EL PROBLEMA DE LA MOCHILA (1)

Este problema surge de distintas maneras. En este capítulo examinaremos la versión más sencilla; en la Sección 8.4 se presentará una variante más difícil.

Nos dan n objetos y una mochila. Para $i = 1, 2,..., n$, el objeto i tiene un peso positivo w_i y un valor positivo v_i. La mochila puede llevar un peso que no sobrepa-

se W. Nuestro objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos transportados, respetando la limitación de capacidad impuesta. En esta primera versión del problema, suponemos que se pueden romper los objetos en trozos más pequeños, de manera que podamos decidir llevar solamente una fracción x_i del objeto i, con $0 \leq x_i \leq 1$. (Si no se nos permite romper los objetos, el problema es mucho más difícil.) En este caso, el objeto i contribuye en $x_i w_i$ al peso total de la mochila, y en $x_i v_i$ al valor de la carga. En símbolos, el problema se puede enunciar en la forma siguiente:

$$\text{maximizar} \sum_{i=1}^{n} x_i v_i \text{ con la restricción } \sum_{i=1}^{n} x_i w_i \leq W$$

donde $v_i > 0$, $w_i > 0$ y $0 \leq x_i \leq 1$ para $1 \leq i \leq n$. Aquí las condiciones que afectan a v_i y a w_i son restricciones sobre el problema; las de x_i son restricciones sobre la solución. Utilizaremos un algoritmo voraz para resolver el problema. En términos de nuestro esquema general, los candidatos son los diferentes objetos, y la solución es un vector $(x_1,...,x_n)$ que nos dice qué fracción de cada objeto hay que incluir. Una solución será factible cuando se respetan las restricciones indicadas anteriormente, y la función objetivo es el valor total de los objetos que están en la mochila. Queda por ver qué debemos tomar como función de selección.

Si $\Sigma_{i=1}^{n} w_i \leq W$, está claro que lo óptimo es meter todos los objetos dentro de la mochila. Por tanto, podemos asumir que en los casos interesantes del problema, es $\Sigma_{i=1}^{n} w_i > W$. También está claro que una solución óptima debe llenar exactamente la mochila, porque en caso contrario se podría añadir una fracción de alguno de los objetos restantes e incrementar el valor de la carga. Por tanto, en una solución óptima $\Sigma_{i=1}^{n} x_i w_i = W$. Puesto que esperamos encontrar un algoritmo voraz que funcione, nuestra estrategia general consistirá en seleccionar cada objeto por turno en algún orden adecuado, poner la mayor fracción posible del objeto seleccionado en la mochila, y detenernos cuando la mochila esté llena. Véase el algoritmo:

```
función mochila(w[1..n], v[1..n], W): matriz [1..n]
    {Inicialización}
    para i = 1 hasta n hacer x[i] ← 0
    peso ← 0
    {bucle voraz}
    mientras peso < N hacer
        i ← el mejor objeto restante {ver más abajo}
        si peso + w[i] ≤ W entonces   x[i] ← 1
                                      peso ← peso + w[i]
                              sino    x[i] ← (W – peso) / w[i]
                                      peso ← W
    devolver x
```

Para este problema hay por lo menos tres funciones de selección posibles: en cada fase, podemos seleccionar el objeto más valioso, argumentando que esto incrementa el valor de la carga del modo más rápido posible; podemos seleccionar el objeto más pequeño restante, basándonos en que de este modo la capacidad se agota de la forma más lenta posible; o bien podemos evitar estos extremos seleccionando aquel objeto cuyo valor por unidad de peso sea el mayor posible. Las figuras 6.5 y 6.6 muestran la forma en que funcionan estas tácticas en un caso particular. Aquí tenemos cinco objetos y $W = 100$. Si seleccionamos los objetos por orden decreciente de valores, entonces seleccionaremos primero el objeto 3, luego el 5 y finalmente llenaremos la mochila con la mitad del objeto 4. El valor de la solución obtenida de esta manera es de 66 + 60 + 40/2 = 146. Si seleccionamos los objetos por orden de peso creciente, entonces seleccionaremos los objetos 1, 2, 3 y 4 por este orden, y la mochila estará llena. El valor de la solución es de 20 + 30 + 66 + 40 = 156. Finalmente, si seleccionamos los objetos por orden decreciente de v_i/w_i, entonces seleccionaremos primero el objeto 3, luego el objeto 1, después el objeto 2, y finalmente llenaremos la mochila con cuatro quintos del objeto 5. Empleando esta táctica, el valor de la solución es 20 + 30 + 66 + 0,8 x 60 = 164.

$n = 5, W = 100$

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

Figura 6.5. Un ejemplar para el problema de la mochila

Este ejemplo muestra que la solución obtenida mediante un algoritmo voraz que maximiza el valor de los objetos seleccionados no es necesariamente óptimo, ni tampoco lo es la solución obtenida minimizando el peso de los objetos seleccionados.

Seleccionar:	x_1					Valor
Máx v_i	0	0	1	0.5	1	146
Mín w_i	1	1	1	1	0	156
Máx v_i/w_i	1	1	1	0	0.8	164

Figura 6.6. Tres enfoques voraces para el ejemplar de la figura 6.5

Afortunadamente la demostración siguiente muestra que la tercera posibilidad, que consiste en seleccionar el objeto que maximice el valor por unidad de peso, lleva realmente a una solución óptima.

Teorema 6.5.1 *Si se seleccionan los objetos por orden decreciente de v_i/w_i, entonces el algoritmo de la mochila encuentra una solución óptima*

Demostración. Supongamos, sin pérdida de generalidad, que los objetos disponibles están ordenados por valor decreciente de coste por unidad de peso, esto es, que

$$v_1 / w_1 \geq w_2 \geq \ldots \geq v_n / w_n$$

Sea $X = (x_1, x_2, \ldots, x_n)$ la solución hallada por el algoritmo voraz. Si todos los x_i son iguales a 1, entonces esta solución es claramente óptima. En caso contrario, supongamos que j denota el menor índice tal que $x_j < 1$. Examinando la forma en que funciona el algoritmo, está claro que $x_i = 1$ cuando $i < j$, que $x_i = 0$ cuando $i > j$, y que $\sum_{i=1}^{n} x_i w_i = W$. Sea $V(X) = \sum_{i=1}^{n} x_i v_i$ el valor de la solución X.

Ahora sea $Y = (y_1, \ldots, y_n)$ cualquier solución factible. Como Y es factible, $\sum_{i=1}^{n} y_i w_i \leq W$ y por tanto $\sum_{i=1}^{n} (x_i - y_i) w_i \geq 0$. Sea $V(Y) = \sum_{i=1}^{n} y_i v_i$ el valor de la solución Y. Ahora

$$V(X) - V(Y) = \sum_{i=1}^{n} (x_i - y_i) v_i = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$$

Cuando $i < j$, $x_i = 1$ y por tanto $x_i - y_i$ es positivo o nulo, mientras que $v_i / w_i \geq v_j / w_j$; cuando $i > j$, $x_i = 0$ y por tanto $x_i - y_i$ es negativo o nulo, mientras que $v_i / w_i \leq v_j / w_j$; y por supuesto cuando $i = j$, $v_i / w_i = v_j / w_j$. Por tanto, en todos los casos se tiene que $(x_i - y_i)(v_i / w_i) \geq (x_j - y_j)(v_j / w_j)$. Por tanto:

$$V(X) - V(Y) \geq (v_j / w_j) \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0$$

Con lo cual, hemos demostrado que ninguna solución factible puede tener un valor mayor que $V(X)$, por lo que la solución X es óptima.

La implementación del algoritmo es directa. Si los objetos ya están ordenados por orden decreciente de v_i/w_i, entonces está claro que el algoritmo voraz requiere un tiempo que está en $O(n)$; el tiempo total incluyendo la ordenación se encuentra por tanto en $O(n \log n)$. Tal como sucedía en la Sección 6.3.1, puede merecer la pena almacenar los objetos en un montículo, con el mayor valor de v_i/w_i en la raíz. La creación del montículo requiere un tiempo que está en $O(n)$, mientras que cada pasada por el bucle voraz requiere un tiempo que está en $O(\log n)$, puesto que la propiedad del montículo debe ser restaurada después de eliminar la raíz. Aun cuando esto no altera el análisis en el caso peor, puede resultar más rápido si sólo se necesitan unos pocos objetos para llenar la mochila.

6.6 PLANIFICACIÓN

En esta sección presentamos dos problemas que conciernen a la forma óptima de planificar tareas en una sola máquina. En el primero, el problema consiste en mi-

nimizar el tiempo medio que invierte cada tarea en el sistema. En el segundo, las tareas tienen un plazo fijo de ejecución, y cada tarea aporta unos ciertos beneficios sólo si está acabada al llegar su plazo: nuestro objetivo es maximizar la rentabilidad. Ambos problemas se pueden resolver empleando algoritmos voraces.

6.6.1 Minimización del tiempo en el sistema

Un único servidor, como por ejemplo un procesador, un surtidor de gasolina, o un cajero de un banco, tiene que dar servicio a n clientes. El tiempo requerido por cada cliente se conoce de antemano: el cliente i requerirá un tiempo t_i para $1 \le i \le n$. Deseamos minimizar el tiempo medio invertido por cada cliente en el sistema. Dado que el número n de clientes está predeterminado, esto equivale a minimizar el tiempo total invertido en el sistema por todos los clientes. En otras palabras, deseamos minimizar

$$T = \sum_{i=1}^{n} (\text{tiempo en el sistema para el cliente } i)$$

Supongamos por ejemplo que tenemos tres clientes, con $t_1 = 5$, $t_2 = 10$ y $t_3 = 3$. Existen seis órdenes de servicio posibles:

Orden	T	
1 2 3:	5 + (5 + 10) + (5 + 10 +3) = 38	
1 3 2:	5 + (5 + 3) + (5 + 3 +10) = 31	
2 1 3:	10 + (10 + 5) + (10 + 5 +3) = 43	
2 3 1:	10 + (10 + 3) + (10 + 3 +5) = 41	
3 1 2:	3 + (3 + 5) + (3 + 5 +10) = 29	← óptimo
3 2 1:	3 + (3 + 10) + (3 + 10 +5) = 34	

En el primer caso, se sirve inmediatamente al cliente 1, el cliente 2 espera mientras se sirve al cliente 1 y entonces le llega el turno, y el cliente 3 espera mientras se sirve a los clientes 1 y 2, y se le sirve en último lugar; el tiempo total invertido en el sistema por los tres clientes es 38. Los cálculos para los demás casos son similares.

En este caso, la planificación óptima se obtiene cuando se sirve a los tres clientes por orden creciente de tiempos de servicio: el cliente 3, que necesita el menor tiempo, es servido en primer lugar, mientras que el cliente 2, que necesita mayor tiempo, es servido en último lugar. Al servir a los clientes por orden decreciente de tiempos de servicio se obtiene la peor planificación. Sin embargo, un ejemplo no es una demostración de que siempre sea así.

Para dar plausibilidad a la idea de que puede ser óptimo planificar los clientes por orden creciente de tiempo de servicio, imaginemos un algoritmo voraz que construye la planificación óptima elemento a elemento. Supongamos que después de planificar el servicio para los clientes $i_1, i_2, \ldots, i_m$ se añade un cliente j. El incre-

mento del tiempo T en esta fase es igual a la suma de los tiempos de servicio para los clientes desde i_1 hasta i_m (porque esto es lo que tiene que esperar el cliente j antes de recibir servicio) más t_j, que es el tiempo necesario para servir al cliente j. Para minimizar esto, dado que un algoritmo voraz nunca reconsidera sus decisiones anteriores, lo único que podemos hacer es minimizar t_j. Nuestro algoritmo voraz, por tanto, es bastante sencillo: en cada paso se añade al final de la planificación al cliente que requiera el menor servicio de entre los restantes.

Teorema 6.6.1 *El algoritmo voraz es óptimo*

Demostración. Sea $P = p_1 p_2 \ldots p_n$ cualquier permutación de enteros del 1 al n, y sea $s_i = t_{p_i}$. Si se sirven clientes en el orden P, entonces el tiempo de servicio requerido por el i-ésimo cliente que haya que servir será s_i, y el tiempo total transcurrido en el sistema por todos los clientes es

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \ldots \\ &= ns_1 + (n-1)s_2 + (n-2)s_3 + \ldots \\ &= \sum_{i=1}^{n} (n - k + 1)s_i \end{aligned}$$

Supongamos ahora que P no organiza a los clientes por orden de tiempos crecientes de servicio. Entonces se pueden encontrar dos enteros a y b con $a < b$ y $s_a > s_b$. En otras palabras, se sirve al a-ésimo cliente antes que al b-ésimo, aun cuando el primero necesite más tiempo de servicio que el segundo; véase la figura 6.7. Si intercambiamos la posición de estos dos clientes, obtendremos un nuevo orden de servicio P' que es simplemente el orden P después de intercambiar los enteros p_a y p_b. El tiempo total transcurrido pasado en el sistema por todos los clientes si se emplea la planificación P' es

$$T(P') = (n - a + 1)s_b + (n - b - 1)s_a + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} (n - k + 1)s_k$$

La planificación nueva es preferible a la vieja, porque

$$\begin{aligned} T(P) - T(P') &= (n - a + 1)(s_a - s_b) + (n - b + 1)(s_b - s_a) \\ &= (b - a)(s_a - s_b) > 0 \end{aligned}$$

Se puede obtener el mismo resultado de manera menos formal a partir de la figura 6.7. Al comparar las planificaciones P y P', se observa que los $a - 1$ primeros clientes salen del sistema exactamente al mismo tiempo en ambas planificaciones. Lo mismo sucede para los $n - b$ últimos clientes. Ahora el cliente a sale cuando antes salía el cliente b, mientras que el cliente b sale antes que salía el cliente a, porque $s_b < s_a$. Finalmente, los clientes que son servidos en las posiciones desde $a + 1$ hasta $b + 1$ también salen antes del sistema, por la misma razón. Por consiguiente, P' es mejor que P en conjunto.

De esta manera, se puede optimizar toda planificación en la que se sirva a un cliente antes que a otro que requiera menos servicio. Las únicas planificaciones que quedan son aquellas que se obtienen poniendo a los clientes por orden no decreciente de tiempo de servicio. Todas estas planificaciones son claramente equivalentes, y por tanto todas son óptimas.

La implementación de este algoritmo es tan sencilla que omitimos los detalles. En esencia, lo único que se necesita es ordenar los clientes por orden de tiempo no decreciente de servicio, lo cual requiere un tiempo que está en $O(n \log n)$. El problema se puede generalizar a un sistema con s servidores, y también el algoritmo: véase el problema 6.20.

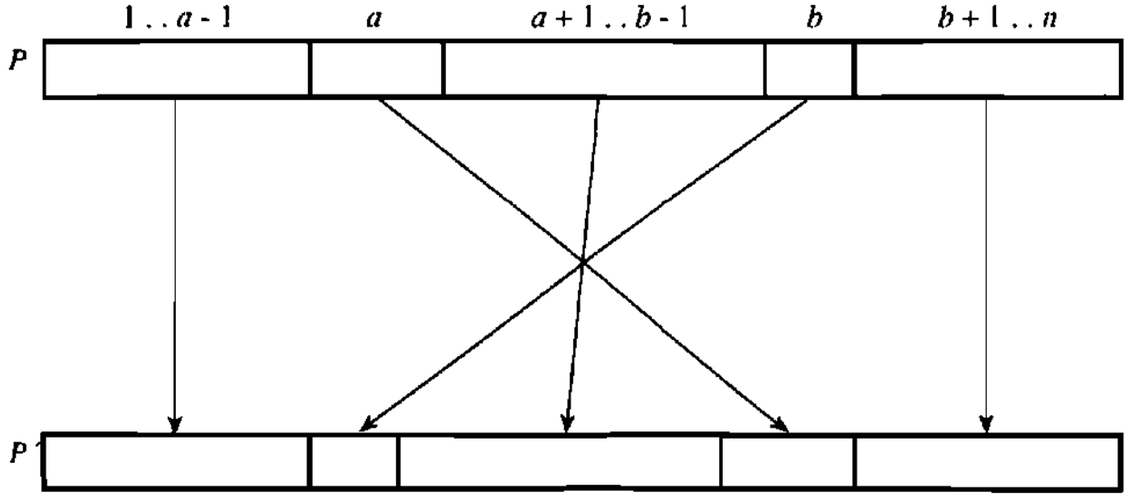


Figura 6.7. Intercambio de dos clientes

6.6.2 Planificación con plazo fijo

Tenemos que ejecutar un conjunto de n tareas, cada una de las cuales requiere un tiempo unitario. En cualquier instante $T = 1, 2, \ldots$ podemos ejecutar únicamente una tarea. La tarea i nos produce unos beneficios $g_i > 0$ sólo en el caso de que sea ejecutada en un instante anterior a d_i.

Por ejemplo, con $n = 4$ y los valores siguientes:

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

las planificaciones que hay que considerar y los beneficios correspondientes son:

Secuencia	Beneficio	Secuencia	Beneficio	
1	50	2,1	60	
2	10	2,3	25	
3	15	3,1	65	
4	30	4,1	80	← óptimo
1,3	65	4,3	45	

Por ejemplo, la secuencia 3,2 no se considera porque la tarea 2 se ejecutaría en el instante $t = 2$, después de su plazo que es $d_2 = 1$. Para maximizar nuestros beneficios en este ejemplo, deberíamos ejecutar la planificación 4,1.

Se dice que un conjunto de tareas es *factible* si existe al menos una sucesión (que también se llama factible) que permite que todas las tareas del conjunto se ejecuten antes de sus respectivos plazos. Un algoritmo voraz evidente consiste en construir la planificación paso a paso, añadiendo en cada paso la tarea que tenga el mayor valor de g_i entre las que aún no se hayan considerado, siempre y cuando el conjunto de tareas seleccionadas siga siendo factible.

En el ejemplo precedente seleccionamos en primer lugar la tarea 1. Después tomamos la tarea 4; el conjunto {1, 4} es factible porque se puede ejecutar en el orden 4, 1. A continuación probamos el conjunto {1, 3, 4}, que resulta no ser factible; por tanto se rechaza la tarea 3. Por último, probamos {1, 2, 4}, que tampoco es factible, así que se rechaza la tarea 2. Nuestra solución (que en este caso es óptima) es por tanto ejecutar el conjunto de tareas {1, 4}, que sólo se puede efectuar en el orden 4, 1. Queda por demostrar que este algoritmo siempre encuentra una planificación óptima, y además hay que buscar una forma eficiente de implementarlo.

Sea J un conjunto de k tareas. A primera vista, puede parecer que necesitamos probar las $k!$ permutaciones de estas tareas para ver si J es factible. Afortunadamente, esto no es así.

Lema 6.6.2 *Sea J un conjunto de k tareas. Supongamos, sin pérdida de generalidad, que las tareas están numeradas de tal forma que $d_1 \leq d_2 \leq \ldots \leq d_k$. Entonces el conjunto J es factible si y sólo si la secuencia $1, 2, \ldots, k$ es factible*

Demostración. El «si» es evidente. Para el «sólo si», supongamos que la secuencia $1, 2, \ldots, k$ no es factible. Entonces al menos una de estas tareas se planifica después de su plazo. Sea r cualquiera de estas tareas, de tal manera que $d_r \leq r - 1$. Dado que las tareas se planifican por orden no decrecientes de plazos, esto significa que al menos r tareas tienen como fecha final $r - 1$ ó anterior. Sea cual fuere la forma en que se planifiquen, la última siempre llegará tarde.

Esto demuestra que basta comprobar una sola secuencia, en orden no decreciente, para saber si un conjunto de tareas J es o no factible.

Teorema 6.6.3 *El algoritmo voraz esbozado anteriormente siempre encuentra una planificación óptima*

Demostración. Supongamos que el algoritmo voraz decide ejecutar un conjunto de tareas I, y supongamos que el conjunto J es óptimo. Sean S_I y S_J secuencias factibles, que posiblemente incluyan huecos, para los dos conjuntos de tareas en cuestión. Reorganizando las tareas de S_I y de S_J, podemos obtener dos secuencias factibles S'_I y S'_J, que también pueden contener huecos, tales que toda tarea común a I y a J se planifique en el mismo instante en ambas secuencias; véase la figura 6.8.

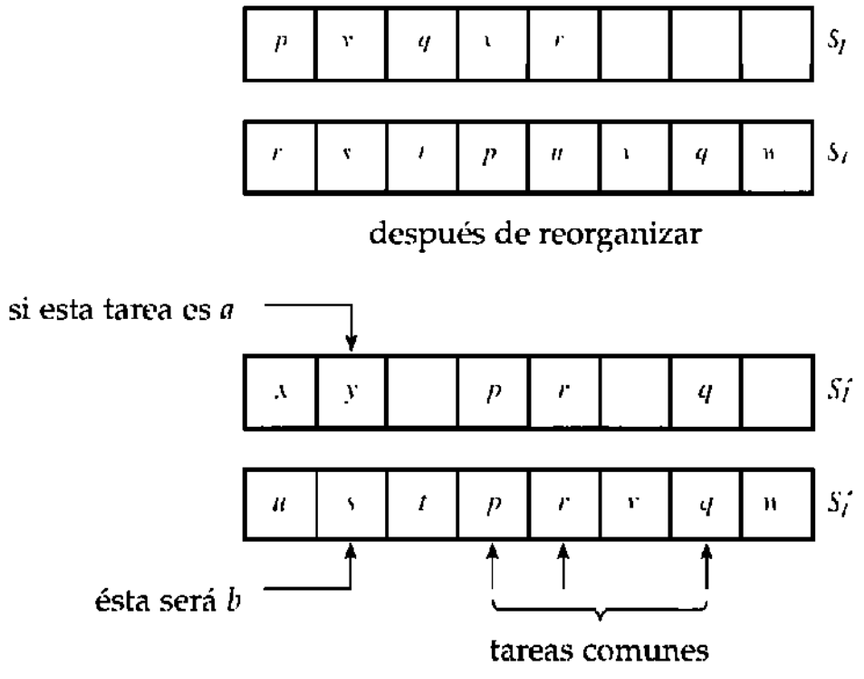


Figura 6.8. Reorganización de tareas para juntar las tareas idénticas

Para ver esto, imaginemos que alguna tarea a aparece en las dos secuencias factibles S_I y S_J, en donde queda planificada en los instantes t_I y t_J respectivamente. Si $t_I = t_J$, no hay nada que hacer. En caso contrario, supongamos que $t_I < t_J$. Dado que la secuencia S_I es factible, se sigue que el plazo para la tarea a no es anterior a t_I. Se modifica la secuencia S_J de la manera siguiente: si hay un hueco en la secuencia S_J en el instante t_I, se atrasa la tarea a del instante t_J al hueco en el instante t_I; si hay una tarea b planificada en S_J en el instante t_I, se intercambian las tareas a y b en la secuencia S_J. La secuencia resultante sigue siendo factible, puesto que en cualquier caso a se ejecutará antes de su plazo, y en el segundo caso el traslado de b

a un instante anterior no puede en modo alguno causar daños. Ahora se planifica a en el mismo instante t_j en las dos secuencias modificadas S_i y S_j. Se puede aplicar un argumento similar cuando $t_i > t_j$, salvo que en este caso es S_j quien debe ser modificada.

Una vez que se ha tratado una tarea a de esta manera, está claro que nunca será preciso volver a trasladarla. Por tanto, si las secuencias S_I y S_J tiene m tareas en común, después de un máximo de m modificaciones de S_I o de S_J podemos asegurar que todas las tareas comunes a I y a J estarán planificadas al mismo tiempo en ambas secuencias. Las secuencias resultantes S'_I y S'_J pueden no ser iguales si $I \neq J$. Por tanto, supongamos que existe un instante en el cual la tarea planificada en S_J es distinta de la planificada en S'_J.

◊ Si alguna tarea a está planificada en S'_I frente a un hueco de S'_J, entonces a no pertenece a J. El conjunto $J \cup \{a\}$ es factible, porque podríamos poner a en el hueco, y sería más rentable que J. Esto es imposible puesto que J es óptimo por hipótesis.

◊ Si alguna tarea b está planificada en S'_J frente a un hueco de S'_I, el conjunto $I \cup \{b\}$ sería factible, así que el algoritmo voraz habría incluido a b en I. Esto también es imposible, porque no lo hizo.

◊ La única posibilidad restante es que alguna tarea a esté planificada en S'_I al lado de una tarea distinta b en S'_J. En este caso, a no aparece en J y b no aparece en I. Aparentemente, hay tres posibilidades:

- Si $g_a > g_b$, se podría sustituir a por b en J y mejorarla. Esto es imposible porque J es óptima.

- Si $g_a < g_b$, el algoritmo voraz habría seleccionado a b antes de considerar a a, puesto que $(I \setminus \{a\}) \cup \{b\}$ sería factible. Esto también es imposible porque el algoritmo no incluyó a b en I.

- La única posibilidad restante es que $g_a = g_b$.

Concluimos que para toda posición temporal las secuencias S'_I y S'_J o bien no planifican tareas, o planifican la misma tarea, o planifican dos tareas distintas que producen idéntico beneficio. El beneficio total de I es por tanto igual al beneficio del conjunto óptimo J, así que I también es óptimo.

Para nuestra primera implementación del algoritmo, supondremos sin pérdida de generalidad que las tareas están numeradas de tal manera que $g_1 \geq g_2 \geq \ldots \geq g_n$. El algoritmo se puede implementar de forma más eficiente (y más sencilla) si suponemos además que $n > 0$ y $d_i > 0$, para $1 \leq i \leq n$, y que está disponible un espacio adicional al principio de los vectores d (que contiene los plazos) y j (en donde se construye la solución). Estas celdas adicionales se conocen con el nombre de «centinelas». Almacenando un valor adecuado en los centinelas se evitan comprobaciones repetitivas de rangos que consumen mucho tiempo.

```
función secuencia(d[0..n]): k, matriz[1..k]
    matriz j[0..n]
    {La planificación se construye paso a paso en la matriz j.
       La variable k dice cuántas tareas están ya
       en la planificación.}
    d[0] ← j[0] ← 0 {centinelas}
    k ← j[1] ← 1 {la tarea 1 siempre se selecciona}
    {bucle voraz}
    para i ← 2 hasta n hacer {orden decreciente de g}
       r ← k
       mientras d[j[r]] > máx(d[i], r) hacer r ← r − 1
       si d[i] > r entonces
       para m ← k paso − 1 hasta r + 1 hacer j[m + 1] ← j[m]
       j[r + 1] ← i
       k ← k + 1
devolver k, j[1..k]
```

Las k tareas de la matriz j están por orden creciente de plazo. Cuando se está considerando la tarea i, el algoritmo comprueba si se puede insertar en j en el lugar oportuno sin llevar alguna tarea que ya esté en j más allá de su plazo. De ser así, i se acepta; en caso contrario, i se rechaza. Los valores exactos de los g_i son innecesarios siempre y cuando las tareas estén numeradas correctamente por orden decreciente de beneficios. La figura 6.9 da g_i y d_i para un ejemplo de seis tareas, y la figura 6.10 ilustra la forma en que funciona el algoritmo en este ejemplo. (La figura 6.10 lo llama algoritmo «lento» porque describiremos uno mejor dentro de poco.)

i	1	2	3	4	5	6
g_i	20	15	10	7	5	3
d_i	3	1	1	3	1	3

Figura 6.9. Un ejemplo con seis tareas

El análisis del algoritmo es sencillo. La ordenación de las tareas por orden decreciente de beneficio requiere un tiempo que está en $\Theta(n \log n)$. El caso peor para el algoritmo es cuando este procedimiento también clasifica las tareas por orden decreciente de plazos, y cuando todas ellas tienen cabida en la planificación. En este caso, cuando se está considerando la tarea i el algoritmo examina las $k = i - 1$ tareas que ya están planificadas, para encontrar un lugar para el recién llegado, y después las desplaza todas un lugar. En términos del programa anterior, hay $\sum_{k=1}^{n-1} k$ pasadas por el bucle **mientras** y $\sum_{m=1}^{n-1} m$ pasadas por el bucle **para** interno. El algoritmo requiere, por tanto, un tiempo que está en $\Omega(n^2)$.

Se obtiene un algoritmo más eficiente cuando se utiliza una técnica distinta para verificar si es factible un conjunto de tareas dado. La nueva técnica depende del lema siguiente.

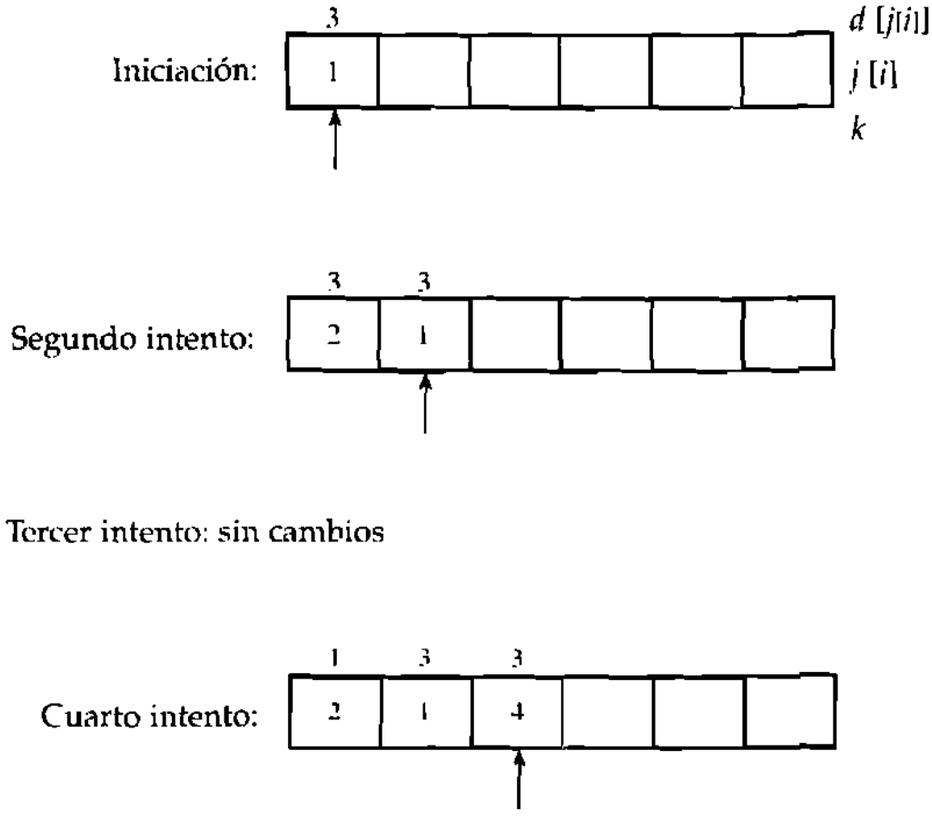


Figura 6.10. Ilustración del algoritmo lento

Lema 6.6.4 Un conjunto J de n tareas es factible si y sólo si se puede construir una secuencia factible que incluya a todas las tareas de J en la forma siguiente. Se empieza con una planificación vacía, de longitud n. Entonces, para cada tarea $i \in J$ sucesivamente, se planifica i en el instante t, en donde t es el mayor entero tal que $1 \leq t \leq \min(n, d_i)$, y la tarea que se ejecutará en el instante t no está decidida todavía.

En otras palabras, se empieza por una planificación vacía, se considera cada tarea sucesivamente, y se añade a la planificación que se está construyendo en el momento más tardío posible, pero no antes de su fecha final. Si no se puede planificar una tarea antes de su plazo, entonces el conjunto J no es factible.

Demostración. El «si» es evidente. Para el « sólo si», obsérvese primero que si existe una secuencia factible, entonces existe una secuencia factible de longitud n. Puesto que sólo hay n taras por planificar, toda secuencia más larga tendrá que contener huecos, y siempre se puede trasladar una tarea a un hueco anterior sin afectar a la factibilidad de la secuencia.

Cuando se intenta añadir una nueva tarea, la secuencia que se está construyendo contiene siempre al menos un hueco. Supongamos que no se puede añadir una tarea cuyo plazo sea d. Esto puede suceder solamente si todas las posiciones desde $t = 1$ hasta $t = r$ están ya reservadas, en donde $r = \min(n, d)$. Sea $s > r$ el menor entero tal que la posición $t = s$ está vacía. La planificación que ya se ha construido incluye por tanto $s - 1$ tareas, ninguna tarea con plazo exactamente s y quizá otras más con plazos posteriores a s. La tarea que estamos intentando añadir también tiene un plazo anterior a s. Por tanto J contiene al menos s tareas cuyas plazos son $s - 1$ o anteriores. Sea cual fuere la forma en que se planifiquen, la última llegará tarde con certeza.

El lema sugiere que deberíamos considerar un algoritmo que intente llenar una por una las posiciones de una secuencia de longitud p, donde $p = min\ (n, max_{1 \le i \le n} di)$. Para cualquier posición t, se define $n_t = \max\{k \le t \mid \text{la posición } k \text{ no está ocupada}\}$. También se definen ciertos conjuntos de posiciones en la forma siguiente: dos posiciones i y j están en el mismo conjunto si $n_i = n_j$; véase la figura 6.11. Para un conjunto dado K de posiciones, sea $F(K)$ el menor elemento de K. Finalmente, se define una posición ficticia cero, que siempre está libre.

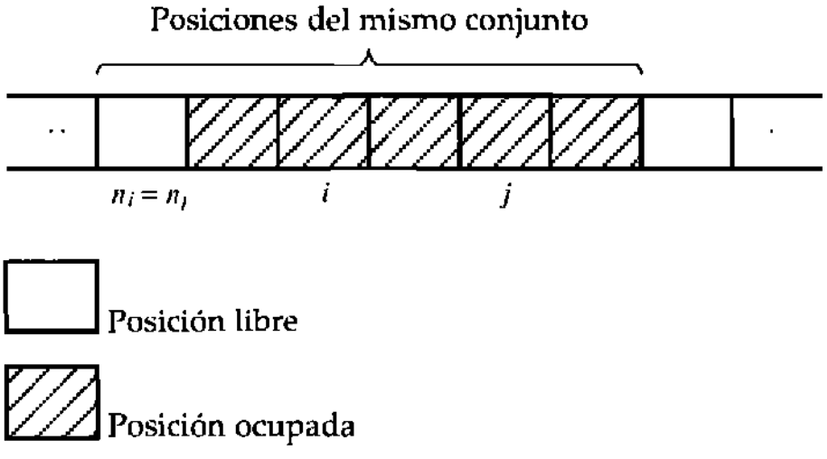


Figura 6.11 Conjuntos de posiciones

A medida que se asignan nuevas tareas a posiciones vacantes, los conjuntos se fusionan para formar conjuntos más grandes: las estructuras de partición están diseñadas precisamente con este propósito. Obtenemos un algoritmo cuyos pasos esenciales son los siguientes:

(*i*) Iniciación: Toda posición 0, 1, 2,..., p está en un conjunto diferente y $F(\{i\}) = i$, $0 \leq i \leq p$.

(ii) Adición de una tarea con plazo d: se busca el conjunto que contenga a d; sea K este conjunto. Si $F(K) = 0$ se rechaza la tarea; en caso contrario:

- Se asigna la nueva tarea a la posición $F(K)$.
- Se busca el conjunto que contenga $F(K) - 1$. Llamemos L a este conjunto (no puede ser igual a K).
- Se fusionan K y L. El valor de F para este nuevo conjunto es el valor viejo de $F(L)$.

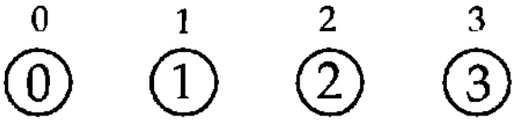
La figura 6.12 ilustra el funcionamiento de este algoritmo para el ejemplo dado en la figura 6.9.

Lo que sigue es un enunciado más preciso del algoritmo rápido. Para simplificar la descripción, suponemos que la etiqueta del conjunto producido por una operación de *fusionar* es necesariamente la etiqueta de uno de los conjuntos que hayan sido fusionados. La planificación producida en primer lugar puede contener huecos; el algoritmo acaba por trasladar tareas hacia adelante para llenarlos:

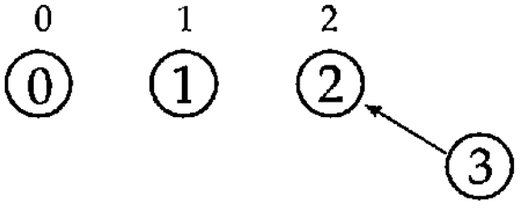
```
función secuencia2(d[1..n]): k; matriz [1..k]
   matriz j, F[0..n]
   {Iniciación}
   p = mín. (n, máx{d[i] | 1 ≤ i ≤ n})
   para i ← 0 hasta p hacer   j[i] ← 0
                              F[i] ← i
                              iniciar el conjunto {i}
   {Bucle voraz}
   para i ← 1 hasta n hacer {orden decreciente de g}
      k ← buscar(mín (p, d[i]))
      m ← F[k]
      si m ≠ 0 entonces
         j[m] ← i
         l ← buscar(m − 1)
         F[k] ← F[l]
         fusionar(k, l) {el conjunto resultante tiene la etiqueta k o l}
   {sólo queda comprimir la solución}
   k ← 0
   para i ← 1 hasta p hacer
      si j[i] > 0 entonces   k ← k + 1
                             j[k] ← j[i]
   devolver k, j[1..k]
```

Si se nos da el problema con las tareas ya ordenadas por beneficios decrecientes, de tal modo que sea posible obtener una secuencia óptima sin más que llamar al algoritmo anterior, entonces la mayor parte del conjunto se invertirá en manipular conjuntos disjuntos. Dado que hay que ejecutar como máximo $2n$ operaciones *buscar* y n operaciones *fusionar*, el tiempo requerido está en $O(n\alpha(2n, n))$, donde α es la función de crecimiento lento de la Sección 5.9. Esto es esencialmente lineal. Si, por otra parte, las tareas se nos dan por un orden arbitrario, entonces tendremos que empezar por ordenarlas, y la obtención de la secuencia inicial requiere un tiempo que está en $\Theta(n \log n)$.

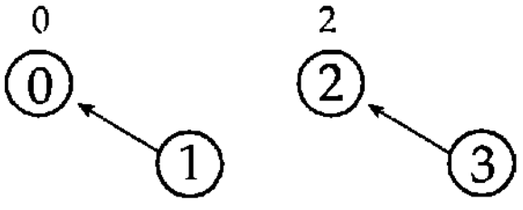
Iniciación: p = mín(6, máx(d_i)) = 3



Intento 1: $d_1 = 3$, se asigna la tarea 1 a la posición 3

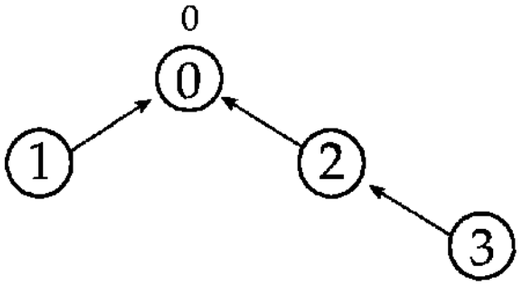


Intento 2: $d_2 = 1$, se asigna la tarea 2 a la posición 1



Intento 3: $d_3 = 1$ no hay posiciones libres disponibles porque el valor de F es 0

Intento 4: $d_4 = 3$, se asigna la tarea 4 a la posición 2



Intento 5: $d_5 = 1$, no hay posiciones libres disponibles

Intento 6: $d_6 = 3$, no hay posiciones libres disponibles

Secuencia óptima: 2, 4, 1; valor = 42

Figura 6.12. Ilustración del algoritmo rápido

6.7 PROBLEMAS

Problema 6.1. ¿Es la ordenación por selección (véase la Sección 2.4) un algoritmo voraz? De ser así, ¿cuáles son las distintas funciones necesarias (la función para comprobar la factibilidad, la función de selección y las demás)?

Problemas 6.2. El sistema monetario inglés, antes de la normalización decimal, constaba de medias coronas (30 peniques), florines (24 peniques), chelines (12 peniques), monedas de seis peniques, tres peniques y peniques (por no mencionar los medios peniques y los *farthings*[3], que valían respectivamente medio y un cuarto de penique respectivamente). Demostrar que con estas monedas el algoritmo voraz de la Sección 6.1 no produce necesariamente una solución óptima, aun cuando se disponga de un suministro inagotable de monedas de cada valor.

Problema 6.3. El sistema monetario portugués consta de monedas de 1, 2'50, 5, 10, 20, 25 y 50 escudos. Sin embargo, los precios son siempre un número entero de escudos. Demostrar lo siguiente, o dar un contraejemplo: cuando se dispone de un suministro inagotable de monedas de cada valor, el algoritmo voraz de la Sección 6.1 siempre encuentra una solución óptima.

Problema 6.4. Supongamos que el sistema monetario consta de los valores dados en la Sección 6.1, pero que nos quedamos sin duros. Mostrar que al utilizar el algoritmo voraz con los valores restantes no se llega necesariamente a una solución óptima.

Problema 6.5. Demostrar lo siguiente, o dar un contraejemplo: supuesto que toda moneda de la serie valga al menos el doble que la moneda de valor inmediatamente inferior, y que la serie contiene una moneda de valor unitario, y se dispone de un suministro inagotable de monedas de cada valor, entonces el algoritmo voraz de la Sección 6.1 siempre encuentra una solución óptima.

Problema 6.6. Supongamos que los valores de monedas disponibles son de la forma $1, p, p^2, \ldots, p^n$ en donde $p > 1$ y $n \geq 0$ son enteros. Demostrar lo siguiente, o dar un contraejemplo: con esta serie de valores y un suministro inagotable de monedas de cada valor, el algoritmo voraz de la Sección 6.1 siempre encuentra una solución óptima.

Problema 6.7. Demostrar que un grafo con n nodos y más de $n - 1$ aristas debe contener al menos un ciclo.

Problema 6.8. Supongamos que el coste de tender una línea telefónica desde el punto a hasta el punto b es proporcional a la distancia euclidiana entre a y b. Es preciso unir entre sí un cierto número de ciudades con el mínimo coste. Buscar un ejemplo en el que cueste menos tender los cables a través de una centralita situada entre ciudades que utilizar solamente enlaces directos de conexión entre ciudades.

Problema 6.9. ¿Qué se puede decir acerca del tiempo requerido por el algoritmo de Kruskal si, en lugar de proporcionar una lista de aristas, el usuario proporciona una matriz de distancias, dejando para el algoritmo la tarea consistente en determinar las aristas que existen?

Problema 6.10. Supongamos que se implementan los algoritmos de Kruskal y el de Prim según se muestra en las Secciones 6.3.1 y 6.3.2 respectivamente. ¿Qué sucede (a) en el caso del algoritmo de Kruskal y (b) en el caso del algoritmo de Prim si por error se ejecuta el algoritmo sobre un grafo que no es conexo?

Problema 6.11. Un grafo puede tener varios árboles de recubrimiento mínimo diferentes. ¿Es esto cierto para el grafo de la figura 6.2? De ser así, ¿dónde se refleja esta posibilidad en los algoritmos explicados en las Secciones 6.3.1 y 6.3.2?

Problema 6.12. El problema consistente en hallar un subconjunto T de las aristas de un grafo conexo G tal que todos los nodos queden conectados utilizando solamente las aristas de T, y cuando la suma de las longitudes de las aristas de T sea la más pequeña posible, sigue teniendo sentido aun cuando G tenga aristas con longitudes negativas. Sin embargo, la solución quizá ya no sea un árbol. Adaptar bien el algoritmo de Kruskal o bien el de Prim para que funcionen en un grafo que pueda incluir aristas de longitud negativa.

Problema 6.13. Demostrar que el algoritmo de Prim, al igual que el algoritmo de Kruskal, se puede implementar utilizando montículos. Demostrar que en tal caso requiere un tiempo que está en $\Theta(a \log n)$.

Problema 6.14. En el algoritmo de Dijkstra, cuando se añade un nuevo nodo v a S, sea w un nodo que no está en S. ¿Es posible que el nuevo camino especial más corto desde el origen hasta w tenga que pasar primero por v y después por algún otro nodo de S?

Problema 6.15. Demostrar, dando un ejemplo explícito, que si las longitudes de las aristas pueden ser negativas, entonces el algoritmo de Dijkstra no siempre funciona correctamente. ¿Sigue teniendo sentido hablar del camino más corto si se admiten distancias negativas?

Problema 6.16. En el análisis de la implementación del algoritmo de Dijkstra que emplea un montículo, se vio que se pueden hacer flotar hasta n nodos, mientras que se eliminan menos de n raíces. La eliminación de la raíz tiene como consecuencia la inmersión del nodo que ocupa su lugar. En general, hacer flotar resulta ligeramente más rápido que sumergir, puesto que en cada nivel se compara el valor del nodo con el valor de su padre, en lugar de hacer comparaciones con los dos hijos. Empleando un montículo k-ario (véase la Sección 5.7 y el Problema 5.23), se puede conseguir que la flotación funcione todavía más deprisa, a costa de hacer más lenta la inmersión. Sea $k = \max(2, \lfloor a / n \rfloor)$. Demostrar la forma de utilizar un montículo k-ario para calcular las rutas más cortas desde un origen hasta todos los demás nodos de un grafo en un tiempo $O(a \log_k n)$. Observe que esto da $O(n^2)$ si $a \approx n^2$ y $O(a \log n)$ si $a \approx n$. Por tanto, nos da lo mejor de ambos mundos.

Problema 6.17. Los montículos de Fibonacci, que se mencionan en la Sección 5.8, tienen las propiedades siguientes. Un montículo que contiene n elementos se puede construir en un tiempo $\Theta(n)$; la búsqueda del mayor elemento, la inserción de un nuevo elemento, el incremento del valor de un elemento y la restauración de la propiedad del montículo, y la fusión de dos montículos requieren todos ellos un tiempo amortizado que está en $\Theta(1)$, y el borrado de cualquier elemento, incluyendo en particular el mayor de ellos, a partir de un montículo que contenga n elementos, requiere un tiempo amortizado que está en $O(\log n)$. Los montículos invertidos de Fibonacci son similares, salvo que las operaciones correspondientes consisten en disminuir el valor de un elemento y buscar o eliminar el elemento más pequeño. Mostrar la forma en que se puede utilizar un montículo invertido de Fibonacci para implementar el algoritmo de Dijkstra en un tiempo que esté en $O(a + n \log n)$.

Problema 6.18. En la Sección 6.5 se suponía que estaban disponibles n objetos, numerados de 1 a n. Supongamos en cambio que tenemos n *tipos* de objetos disponibles, con un suministro adecuado de objetos de cada tipo. Formalmente, esto sólo cambia la restricción anterior $0 \leq x_i \leq 1$ por la restricción más permisiva $x_i \geq 0$. ¿Sigue funcionando el algoritmo voraz de la Sección 6.5? ¿Sigue siendo necesario?

Problema 6.19. Demostrar lo siguiente o dar un contraejemplo: para el problema de planificación con plazos de la Sección 6.6.1, la planificación de clientes por orden descendente de tiempos de servicio da lugar a la peor planificación posible.

Problema 6.20. Como en la Sección 6.6.1, tenemos n clientes. El cliente i, para $1 \leq i \leq n$, requiere un tiempo de servicio conocido t_i. Sin pérdida de generalidad, supongamos que los clientes están numerados de tal manera que $t_1 \leq t_2 \leq \ldots \leq t_n$. Si hay s servidores idénticos, demostrar que para minimizar el tiempo total (y por tanto el tiempo medio) pasado en el sistema por los clientes, el servidor j, $1 \leq j \leq s$, debe dar servicio a los clientes j, $j + s$, $j + 2s$... por ese orden.

Problema 6.21. Sean n programas P_1, P_2, ..., P_n que hay que almacenar en un disco. El programa P_i requiere s_i kilobytes de espacio, y la capacidad del disco es D kilobytes, donde $D < \Sigma_{i=1}^{n} s_i$.

(a) Se desea maximizar el número de programas almacenados en el disco. Demostrar lo siguiente, o dar un contraejemplo: podemos utilizar un algoritmo voraz que seleccione los programas por orden no decreciente de s_i.

(*b*) Se desea utilizar la mayor capacidad posible del disco. Demostrar lo siguiente o dar un contraejemplo: se puede utilizar un algoritmo voraz que seleccione los programas por orden no creciente de s_i.

Problema 6.22. Sean n programas P_1, P_2,... P_n que hay que almacenar en una cinta. El programa P_i requiere s_i kilobytes de espacio; la tapa es suficientemente larga para almacenar todos los programas. Sabemos con qué frecuencia se utiliza cada programa: una fracción π_i de las solicitudes afecta al programa i (y por tanto $\Sigma_{i=1}^{n} \pi_i = 1$). La información en la cinta es almacenada (grabada) con densidad constante, y la velocidad de la cinta también es constante. Una vez que se carga el programa, la cinta se rebobina hasta el principio. Si los programas se almacenan por orden $i_1, i_2, \ldots, i_n$ el tiempo medio requerido para cargar un programa es, por tanto:

$$\bar{T} = c \sum_{l=1}^{n} \left[\pi_l \sum_{k=1}^{l} s_{i_k} \right]$$

donde la constante c depende de la densidad de grabación y de la velocidad de la cinta. Deseamos minimizar $\bar{T}$ empleando un algoritmo voraz. Demostrar todo lo siguiente, o dar un contraejemplo: podemos seleccionar los programas (a) por orden de s_i no decreciente; (*b*) por orden de π_i no creciente; (*c*) por orden de π_i / s_i no creciente.

Problema 6.23. Supongamos que las dos planificaciones S_I y S_J presentadas en la demostración de optimalidad de la Sección 6.6.2 se dan en la forma de matrices $SI[1..r]$ y $SJ[1..r]$, donde $r = \max_{1 \leq i \leq n} d_i$. Un elemento de la matriz contiene i si la tarea i debe ejecutarse en el momento correspondiente, y cero representa un hueco en la planificación. Escribir un algoritmo que produzca las planificaciones S_I' y S_J' en las matrices SI y SJ respectivamente.

6.8 REFERENCIAS Y TEXTOS MÁS AVANZADOS

Edmonds (1971) introdujo la noción de algoritmo voraz. Se puede encontrar una descripción del algoritmo voraz para dar el cambio en Wright (1975), y en Chang y Korsh (1976). Volveremos a este problema en la Sección 8.2, lugar en el que proporcionamos un algoritmo más lento, pero que garantiza devolver la solución óptima en todos los casos.

El problema de los árboles de recubrimiento (de envergadura) mínimo tiene una larga historia, que se describe en Graham y Hell (1985). El primer algoritmo propuesto (que no hemos descrito) se debe a Borůvka (1926). El algoritmo al que se asocia el nombre de Prim fue inventado por Jarník (1930) y redescubierto por Prim (1957) y Dijkstra (1959). El algoritmo de Kruskal proviene de Kruskal (1956). Otros algoritmos más sofisticados son los dados por Yao (1975), Cheriton y Tarjan (1976), y Tarjan (1983); véase también el párrafo siguiente.

La implementación del algoritmo de Dijkstra que requiere un tiempo que está en $O(n^2)$ es de Dijkstra (1959). Los detalles de la mejora sugerida en el Problema 6.16, que utiliza montículos *k-arios*, se puede encontrar en Johnson (1977). Una mejora similar para el problema de árbol de recubrimiento mínimo ha sido sugerida por Johnson (1975). Se dan algoritmos más rápidos para estos dos problemas en Fredman y Tarjan (1987); en particular, el uso del montículo de Fibonacci les permite implementar el algoritmo de Dijkstra en un tiempo que está en $O(a + n \log n)$. Se pueden encontrar otras ideas acerca de las rutas más cortas en Tarjan (1983).

La solución del Problema 6.8 usa la noción de árboles de Steiner. El problema de buscar un árbol de Steiner mínimo es de dificultad NP —véase la Sección 12.5.5— y por tanto es probablemente mucho más difícil que hallar un árbol de recubrimiento mínimo. Para más información acerca de este problema, véase Garey, Graham y Johnson (1977) y Winter (1987).

Un algoritmo voraz importante y que no hemos discutido se utiliza para derivar códigos de Huffman óptimos; véase Schwartz (1964). Otros algoritmos voraces para toda una gama de problemas se describen en Horowitz y Sahni (1978).

Capítulo 7

Divide y vencerás

Divide y vencerás es una técnica para diseñar algoritmos que consiste en descomponer el caso que haya que resolver en un cierto número de subcasos más pequeños del mismo problema, resolver sucesiva e independientemente todos estos subcasos, y combinar después las soluciones obtenidas de esta manera para obtener la solución del caso original. Las dos preguntas que surgen de modo natural son: ¿por qué querría nadie hacer esto? y ¿cómo se resuelven los subcasos? La eficiencia de la técnica de divide y vencerás se debe a la respuesta a esta última pregunta.

7.1 INTRODUCCIÓN: MULTIPLICACIÓN DE ENTEROS MUY GRANDES

Consideremos una vez más el problema de multiplicar enteros muy grandes. Recordemos que el algoritmo clásico (figura 1.1) que aprendemos en la escuela requiere un tiempo en $\Theta(n^2)$ para multiplicar números de n cifras. Estamos tan acostumbrados a este algoritmo que quizá nunca haya tenido dudas acerca de su optimalidad. ¿Podemos hacerlo mejor? La multiplicación *à la russe* (figura 1.2) no ofrece mejoras del tiempo de ejecución. Otro algoritmo que discutíamos en la Sección 1.2, y que llamábamos técnica de «divide y vencerás» (figura 1.3) consistía en reducir la multiplicación de dos números de n cifras a cuatro multiplicaciones de números de $\frac{n}{2}$ cifras. Lamentablemente, el algoritmo resultante no proporciona ninguna mejora con respecto al algoritmo clásico de multiplicación, a no ser que seamos más inteligentes. Para superar al algoritmo clásico, debemos encontrar una forma de reducir la multiplicación original no a cuatro sino a *tres* multiplicaciones de números de tamaño mitad.

Ilustraremos el proceso con el ejemplo utilizado en la Sección 1.2: la multiplicación de 981 por 1.234. En primer lugar se rellena el operando más corto con un cero no significativo para hacerlo de longitud igual a la del más largo; de esta manera 981 se convierte en 0981. Entonces partimos en dos mitades ambos operandos: 0981 da lugar a $w = 09$ y $x = 81$, y 1.234 da lugar a $y = 12$ y $z = 34$. Obsérvese que $981 = 10^2w + x$ y que $1.234 = 10^2y + z$. Por tanto, el producto requerido se puede calcular en la forma siguiente:

$$\begin{aligned} 981 \times 1.234 &= (10^2w + x) \times (10^2y + z) \\ &= 10^4wy + 10^2(wz + xy) + xz \\ &= 1.080.000 + 127.800 + 2.754 = 1.210.554 \end{aligned}$$

Si piensa que no hemos hecho otra cosa que reescribir el algoritmo de la Sección 1.2 con más símbolos, estará en lo cierto. El procedimiento anterior sigue necesitando cuatro multiplicaciones de números de tamaño mitad: wy, wz, xy y xz.

La observación clave es que no hay necesidad de calcular wz y xy; lo único que necesitamos realmente es la *suma* de estos dos términos. ¿Es posible obtener $wz + xy$ a costa de una sola multiplicación? Esto parece imposible hasta que recordamos que también se necesitan los valores de wy y de xz para aplicar la fórmula anterior. Teniendo esto en cuenta, consideremos el producto

$$r = (w + x) \times (y + z) = wy + (wz + xy) + xz$$

Al cabo de una sola multiplicación, obtenemos la suma de los tres términos necesarios para calcular el producto deseado. Esto sugiere que procedamos en la forma siguiente:

$$\begin{aligned} p &= wy = 09 \times 12 && = 108 \\ q &= xz = 81 \times 34 && = 2.754 \\ r &= (w + x) \times (y + z) && = 90 \times 46 = 4.140 \end{aligned}$$

Y finalmente

$$\begin{aligned} 981 \times 1.234 &= 10^4p + 10^2(r - p - q) + q \\ &= 1.080.000 + 127.800 + 2.754 = 1.210.554 \end{aligned}$$

De esta manera, el producto de 981 por 1.234 se puede reducir a *tres* multiplicaciones de números de dos cifras (09×12, 81×34 y 90×46) junto con un cierto número de desplazamientos (multiplicaciones por potencias de 10), adiciones y sustracciones.

Desde luego, el número de sumas —contando las restas como si fueran sumas— es mayor que en el algoritmo divide y vencerás original de la Sección 1.2. ¿Merece la pena efectuar cuatro sumas más para ahorrar una multiplicación? La respuesta es negativa cuando se multiplican números pequeños como los de nuestro ejemplo. Sin embargo, merece la pena cuando los números que hay que multiplicar son grandes, y cada vez más cuanto mayores sean los números. Cuando los operandos son grandes, el tiempo requerido para las sumas es despreciable frente al tiempo que requiere una sola multiplicación. Entonces parece razonable esperar que la reducción de cuatro multiplicaciones a tres nos permita reducir en un 25% el tiempo requerido para grandes multiplicaciones. Como veremos, nuestro ahorro va a ser significativamente mayor.

Para ayudarnos a comprender lo que hemos logrado, supongamos que una implementación dada del algoritmo clásico de multiplicación requiere un tiempo

$h(n) = cn^2$ para multiplicar dos números de n cifras, para alguna constante c que dependerá de la implementación (esto es una simplificación, porque en realidad el tiempo requerido tendría una forma más complicada, tal como $cn^2 + bn + a$). De manera similar, sea $g(n)$ el tiempo requerido por el algoritmo divide y vencerás para multiplicar dos números de n cifras, *sin* contar el tiempo necesario para efectuar las tres multiplicaciones de números de tamaño mitad. En otras palabras, $g(n)$ es el tiempo necesario para las sumas, desplazamientos y operaciones adicionales varias. Resulta sencillo implementar estas operaciones de tal manera que $g(n) \in \Theta(n)$. Ignore por el momento lo que sucede si n es impar o si los números no son de la misma longitud.

Si las tres multiplicaciones de números de tamaño mitad se efectúan mediante el algoritmo clásico, el tiempo necesario para multiplicar los dos números de n cifras es:

$$3h(n/2) + g(n) = 3c(n/2)^2 + g(n) = \tfrac{3}{4} cn^2 + g(n) = \tfrac{3}{4} h(n) + g(n)$$

Dado que $h(n) \in \Theta(n^2)$ y $g(n) \in \Theta(n)$, el término $g(n)$ resulta despreciable en comparación con $\frac{3}{4}h(n)$ cuando n es suficientemente grande, lo cual significa que hemos ganado aproximadamente un 25% de velocidad en comparación con el algoritmo clásico, tal como anticipábamos. Aun cuando esta mejora no es despreciable, no hemos conseguido modificar el orden del tiempo requerido: el nuevo algoritmo sigue requiriendo un tiempo cuadrático.

Para mejorar esto, volvemos a la cuestión planteada en el párrafo inicial: ¿cómo hay que resolver los subejemplares? Si son pequeños, es posible que el algoritmo clásico siga siendo la mejor manera de proceder. Sin embargo, cuando los subejemplares sean suficientemente grandes, ¿no sería mejor utilizar nuestro nuevo algoritmo de *forma recursiva*? ¡La idea es análoga a beneficiarse de una cuenta bancaria que componga los pagos de interés! Cuando hacemos esto, obtenemos un algoritmo que puede multiplicar dos números de n cifras en un tiempo $t(n) = 3t(n/2) + g(n)$ cuando n es par y suficientemente grande. Esta recurrencia es parecida a la que estudiábamos en la Sección 4.7.1, y al ejemplo 4.7.10; al resolverla se obtiene que $t(n) \in \Theta(n^{\lg 3} \mid n$ es una potencia de 2). Tenemos que contentarnos con nuestra notación asintótica condicional porque todavía no hemos atacado la cuestión de cómo se multiplican números de longitud impar; véase el Problema 7.1.

Dado que $\lg 3 \approx 1{,}585$ es menor que 2, este algoritmo puede multiplicar dos enteros grandes mucho más deprisa que el algoritmo clásico de multiplicación, y cuanto mayor sea n, más merecerá la pena esta mejora. Es probable que una buena implementación no utilice la base 10, si no más bien la mayor de las bases para las que el *hardware* admita multiplicar directamente dos dígitos. Recuerde que el rendimiento de este algoritmo y el del algoritmo clásico se han comparado empíricamente al final de la Sección 2.7.3.

Un factor importante para la eficiencia práctica de este algoritmo de multiplicación, y de hecho de cualquier algoritmo del tipo divide y vencerás, consiste en saber cuándo hay que dejar de dividir los casos para empezar a utilizar directamen-

te el algoritmo clásico. Aun cuando el enfoque de divide y vencerás es mejor cuanto más grande es el caso, lo cierto es que puede resultar *más lento* que el algoritmo clásico en casos que sean demasiado pequeños. Por tanto, un algoritmo de divide y vencerás debe de evitar seguir avanzando recursivamente cuando el tamaño de los casos ya no lo justifique. Volveremos a este asunto en la sección siguiente.

Por sencillez, hay varios asuntos importantes que han quedado en el tintero hasta este momento. ¿Cómo se tratan los números de longitud impar? Aun cuando las dos mitades del multiplicando y del multiplicador sean de tamaño $n/2$, puede suceder que su *suma* se desborde, y sea de un tamaño una unidad mayor. Por tanto, era ligeramente incorrecto afirmar que $r = (w + x) \times (y + z)$ necesita una multiplicación de números de tamaño mitad. ¿Cómo afecta esto al análisis del tiempo de ejecución? ¿Cómo multiplicamos dos números de diferentes tamaños? ¿Existen otras operaciones aritméticas, además de la multiplicación, que podamos manejar más eficientemente que mediante el uso de los algoritmos clásicos?

Los números de longitud impar se multiplican fácilmente partiéndolos del modo más equitativo posible: un número de n cifras se parte en un número de $\lfloor n/2 \rfloor$ cifras y otro de $\lceil n/2 \rceil$ cifras. La segunda cuestión es más peliaguda. Consideremos la multiplicación de 5.678 por 6.789. Nuestro algoritmo parte los operandos en la forma $w = 56$, $x = 78$, $y = 67$ y $z = 89$. Las tres multiplicaciones de tamaño mitad implicadas son:

$$p = wy = 56 \times 67$$
$$q = xz = 78 \times 69, y$$
$$r = (w + x) \times (y + z) = 134 \times 156$$

La tercera multiplicación usa números de *tres* cifras, y por tanto no son realmente de tamaño mitad en comparación con los números originales de cuatro cifras. Sin embargo, el tamaño de $w + x$ y el de $y + z$ no puede sobrepasar $1 + \lceil n/2 \rceil$.

Para simplificar el análisis, supongamos que $t(n)$ denota el tiempo requerido por este algoritmo en el caso peor para multiplicar dos números cuyo tamaño no sea mayor que n (en lugar de ser exactamente n). Por definición, $t(n)$ es una función no decreciente. Cuando n es suficientemente grande, nuestro algoritmo reduce la multiplicación de dos números de tamaño menor o igual que n a tres multiplicaciones $p = wy$, $q = xz$ y $r = (w + x) \times (y + x)$, de números de tamaños que son menores o iguales que $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ y $1 + \lceil n/2 \rceil$, respectivamente, además de algunas manipulaciones sencillas que requieren un tiempo en $O(n)$. Por tanto, existe una constante real y positiva c tal que

$$t(n) \leq t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + t(1 + \lceil n/2 \rceil) + cn$$

para todo n suficientemente grande. Esta es precisamente la recurrencia estudiada en el ejemplo 4.7.14, y que da lugar al ya familiar $t(n) \in O(n^{\lg 3})$. Por tanto, siempre es posible multiplicar números de n cifras en un tiempo que está en $O(n^{\lg 3})$. Un análisis de este algoritmo en el caso peor muestra que de hecho $t(n) \in \Theta(n^{\lg 3})$, pero es-

te resultado tiene un interés limitado porque existen algoritmos de multiplicación todavía más rápidos; véanse los problemas 7.2 y 7.3.

Volviendo a la cuestión de multiplicar números de distintos tamaños, sean u y v dos enteros de tamaños m y n respectivamente. Si m y n no difieren en más de un factor 2, lo mejor es rellenar el operando más pequeño con ceros no significativos, para hacerlo de longitud igual a la del otro operando, tal como se hacía al multiplicar 981 por 1.234. Sin embargo, este enfoque no resulta recomendable cuando uno de los operandos es mucho mayor que el otro. ¡Podría ser incluso peor que utilizar el algoritmo clásico de multiplicación! Sin pérdida de generalidad, supongamos que $m \leq n$. El algoritmo divide y vencerás utilizado con relleno y el algoritmo clásico requieren un tiempo que está en $\Theta(n^{\lg 3})$, y en $\Theta(mn)$ respectivamente, para calcular el producto de u y v. Considerando que la constante oculta del primero tiene probabilidades de ser más grande que la del segundo, vemos que el algoritmo divide y vencerás con relleno es *más lento* que el algoritmo clásico cuando $m < n^{\lg(3/2)}$, y en particular cuando $m < \sqrt{n}$.

Sin embargo, es fácil combinar los dos algoritmos para obtener un algoritmo realmente mejor. La idea es segmentar el operando más largo, v, en bloques de tamaño m, y utilizar entonces el algoritmo divide y vencerás para multiplicar u por cada bloque de v, de tal manera que se utiliza el algoritmo divide y vencerás para multiplicar parejas de operandos de igual tamaño. El producto final de u y v se obtiene entonces fácilmente mediante adiciones y desplazamientos sencillos. El tiempo total de ejecución está dominado por la necesidad de efectuar $\lceil n / m \rceil$ multiplicaciones de números de m cifras. Dado que cada una de estas multiplicaciones más pequeñas requiere un tiempo que está en $\Theta(m^{\lg 3})$, y teniendo en cuenta que $\lceil n / m \rceil \in \Theta(n / m)$, el tiempo total de ejecución necesario para multiplicar un número de n cifras por un número de m cifras está en $\Theta(nm^{\lg(3/2)})$ cuando $m \leq n$.

La multiplicación no es la única operación interesante que utiliza enteros grandes. La exponenciación modular también es crucial para la criptografía moderna; véase la Sección 7.8. La división entera, las operaciones de módulo y el cálculo de la parte entera de una raíz cuadrada se pueden efectuar todas ellas en un tiempo cuyo orden es el mismo que se requiere para la multiplicación; véase la Sección 12.4. Otras operaciones importantes, tales como el cálculo del máximo común divisor, pueden ser inherentemente más difíciles de calcular y no se tratarán aquí.

7.2 EL CASO GENERAL

La multiplicación de enteros muy grandes no es un ejemplo aislado del beneficio que se puede obtener empleando el enfoque de divide y vencerás. Consideremos un problema arbitrario, y sea *ad hoc* un algoritmo sencillo capaz de resolver el problema. De *ad hoc* nos interesa que sea eficiente para casos pequeños, pero su rendimiento para casos grandes no nos concierne. Lo llamamos *subalgoritmo básico*. El algoritmo clásico de multiplicación es un ejemplo de subalgoritmo básico.

El caso general de los algoritmos de divide y vencerás es como sigue:

función $DV(x)$
  si x es suficientemente pequeño o sencillo, **entonces proporcionar** *ad hoc*(x)
  descomponer x en casos más pequeños $x_1, x_2, \ldots, x_\ell$
  para $i \leftarrow 1$ **hasta** ℓ **hacer** $y_i \leftarrow DV(x_i)$
  recombinar los y_i para obtener una solución y de x
  devolver y

Algunos algoritmos de divide y vencerás no siguen exactamente este esquema: por ejemplo, pueden necesitar que el primer subejemplar esté resuelto antes de formular el segundo subejemplar; véase la Sección 7.5.

El número de subejemplares, ℓ, suele ser pequeño e independiente del caso particular que haya que resolver. Cuando $\ell = 1$, no tiene mucho sentido «*descomponer* x en un caso más sencillo x_1», y es difícil justificar el nombre de *divide y vencerás* que se le da a esta técnica. Sin embargo, tiene sentido reducir la resolución de un caso muy grande a la de uno más pequeño. Entonces, divide y vencerás recibe el nombre de *reducción (simplificación)*; véanse las Secciones 7.3 y 7.7. Cuando se utiliza la simplificación, a veces es posible sustituir la recursividad inherente a divide y vencerás por un bucle iterativo. Cuando se implementa en un lenguaje convencional como Pascal y en una máquina convencional que utilice un compilador poco sofisticado, un algoritmo iterativo tiene probabilidades de ser más rápido que la versión recursiva, aunque sólo por una constante multiplicativa. Por otra parte, puede ser posible ahorrar una cantidad considerable de memoria de esta manera: para un caso de tamaño n, el algoritmo recursivo utiliza una pila cuyo tamaño está en $\Omega(\lg n)$, y en casos malos incluso en $\Omega(n)$.

Para que el enfoque divide y vencerás merezca la pena, es necesario que se cumplan tres condiciones. La decisión de utilizar el subalgoritmo básico en lugar de hacer llamadas recursivas debe tomarse cuidadosamente; tiene que ser posible descomponer el ejemplar en subejemplares y recomponer las soluciones parciales de forma bastante eficiente, y los subejemplares deben ser en lo posible aproximadamente del mismo tamaño. La mayoría de los algoritmos de divide y vencerás son tales que el tamaño de los ℓ subejemplares es aproximadamente n/b para alguna constante b, en donde n es el tamaño del caso original. Por ejemplo, nuestro algoritmo de divide y vencerás para multiplicar enteros muy grandes requiere un tiempo en $\Theta(n)$ para descomponer el caso original en tres subejemplares de tamaños próximos a la mitad, y para recombinar las soluciones parciales: $\ell = 3$ y $b = 2$.

El análisis de tiempos de ejecución para estos algoritmos de divide y vencerás es casi automático, gracias a los ejemplos 4.7.13 y 4.7.16. Sea $g(n)$ el tiempo requerido por DV en casos de tamaño n, sin contar el tiempo necesario para llamadas recursivas. El tiempo total $t(n)$ requerido por este algoritmo de divide y vencerás es parecido a:

$$t(n) = lt(n \div b) + g(n)$$

siempre que n sea suficientemente grande. Si existe un entero k tal que $g(n) \in \Theta(n^k)$, el ejemplo 4.7.16 es aplicable para concluir que:

$$t(n) \in \begin{cases} \Theta(n^k) & \text{si } \ell < b^k \\ \Theta(n^k \log n) & \text{si } \ell = b^k \\ \Theta(n^{\log_b \ell}) & \text{si } \ell > b^k. \end{cases} \tag{7.1}$$

Las técnicas utilizadas en la Sección 4.7.6 y en el ejemplo 4.7.14 son aplicables en general para obtener la misma conclusión aun cuando algunos de los subejemplares sean de un tamaño que sea distinto de $\lfloor n/b \rfloor$ en una constante aditiva, y en particular si algunos de los subejemplares son de tamaño $\lceil n/b \rceil$. Como ejemplo, nuestro algoritmo de divide y vencerás para la multiplicación de grandes enteros se caracteriza por $\ell = 3$, $b = 2$ y $k = 1$. Dado que $\ell > b^k$, es aplicable el tercer caso y obtenemos inmediatamente que el algoritmo requiere un tiempo que está en $\Theta(n^{\lg 3})$ sin tener que preocuparnos por el hecho de que dos de los subejemplares son de tamaños $\lceil n/2 \rceil$ y $1 + \lceil n/2 \rceil$ en lugar de $\lfloor n/2 \rfloor$. En situaciones más complicadas, en las que $g(n)$ no es del orden exacto de un polinomio, puede ser aplicable el Problema 4.44.

Queda por ver la forma de determinar si hay que dividir el caso y hacer llamadas recursivas, o si el caso es tan sencillo que resulta mejor invocar directamente al subalgoritmo básico. Aun cuando esta decisión no afecta al orden del tiempo de ejecución del algoritmo, nos interesa que la constante multiplicativa oculta en la notación Θ sea lo más pequeña posible. En la mayoría de los algoritmos de divide y vencerás, esta decisión se basa en un sencillo *umbral*, que suele denotarse mediante n_0. El subalgoritmo básico se emplea para resolver todos aquellos casos cuyo tamaño no supere n_0.

Volvamos al problema de multiplicar enteros muy grandes, para ver por qué es importante la selección de un umbral, y la forma de seleccionarlo. Para evitar oscurecer las cuestiones esenciales, utilizaremos una fórmula de recurrencia simplificada para el tiempo de ejecución del algoritmo divide y vencerás para multiplicar enteros grandes:

$$t(n) = \begin{cases} h(n) & \text{si } n \le n_0 \\ 3t(\lceil n/2 \rceil) + g(n) & \text{en caso contrario} \end{cases}$$

en donde $h(n) \in \Theta(n^2)$ y $g(n) \in \Theta(n)$.

Consideremos, por ejemplo, una implementación en la cual $h(n) = n^2$ y $g(n) = 16n$ microsegundos. Supongamos que tenemos que multiplicar dos números de 5.000 cifras. Si el algoritmo divide y vencerás procede de forma recursiva hasta obtener subejemplares de tamaño 1, esto es, si $n_0 = 1$, entonces se necesitan más de 41 segundos para calcular el producto. Esto es ridículo, puesto que estos dos números se pueden multiplicar en 25 segundos empleando el algoritmo clásico. El algoritmo clásico es ligeramente mejor que el algoritmo divide y vencerás incluso para multiplicar números de 32.789 cifras, caso en el que ambos algoritmos requieren más de un cuarto de

hora de tiempo de cálculo ¡para efectuar una única multiplicación! ¿Debemos concluir que el algoritmo de divide y vencerás nos permite pasar de un algoritmo cuadrático a uno cuyo tiempo de ejecución está en $\Theta(n^{\lg 3})$, pero sólo a costa de un incremento tan enorme de la constante oculta que el nuevo algoritmo nunca resulta económico en casos de tamaño razonable? Afortunadamente, no es así: para seguir con nuestro ejemplo, se pueden multiplicar números de 5.000 cifras en poco más de 6 segundos, siempre y cuando seleccionemos de forma inteligente el umbral n_0; en este caso, $n_0 = 64$ es una buena opción. Con el mismo umbral, se necesita poco más de dos minutos para multiplicar dos números de 32.789 cifras.

La selección del mejor umbral se ve complicada por el hecho de que el valor óptimo depende en general no sólo del algoritmo en cuestión, sino también de la implementación particular. Además, no existe en general un valor uniformemente mejor del umbral. En nuestro ejemplo, lo mejor es utilizar el algoritmo clásico para multiplicar números de 67 cifras, mientras que es mejor utilizar la recurrencia una sola vez para multiplicar números de 66 cifras. Por tanto, 67 es mejor que 64 como umbral en el primer caso, mientras que en el segundo es mejor lo contrario. En lo sucesivo, emplearemos el término «umbral óptimo» para indicar *casi óptimo.*

¿Y entonces, qué seleccionamos como umbral? Dada una implementación particular, el umbral óptimo se puede determinar empíricamente. Variamos el valor del umbral y el tamaño de los casos usados para nuestra prueba, y cronometramos la implementación en cierto números de casos. Suele ser posible estimar un umbral óptimo tabulando los resultados de estas pruebas, o incluso dibujando unos cuantos diagramas. Sin embargo, los cambios del umbral dentro de un cierto intervalo pueden no tener efecto sobre la eficiencia del algoritmo cuando sólo se consideren casos de un tamaño específico. Por ejemplo, se necesita exactamente el mismo tiempo para multiplicar dos números de 5.000 cifras cuando se le da al umbral cualquier valor entre 40 y 78, puesto que cualquiera de estos valores para el umbral hace que la recursividad se detenga cuando los subcasos llegan al tamaño 40, bajando desde el tamaño 79, en el séptimo nivel de recursión. Sin embargo, estos umbrales no son equivalentes en general, porque los números de 41 cifras requieren un 17% más de tiempo para multiplicarlos con el umbral 40 frente al umbral 64. Por tanto, lo normal es que no baste con variar sencillamente el umbral para un caso de tamaño fijo. La aproximación empírica puede requerir unas cantidades notables de tiempo de computadora (¡y de tiempo humano!). En cierta ocasión, pedimos a los alumnos de un curso de algoritmia que implementasen el algoritmo divide y vencerás para multiplicar enteros muy grandes, con objeto de compararlo con el algoritmo clásico. Varios grupos intentaron estimar empíricamente el umbral óptimo, y cada grupo invirtió en el intento ¡más de 5.000 dólares de tiempo de computadora! Por otra parte, un cálculo puramente teórico del umbral óptimo no suele ser posible, dado que varía de unas implementaciones a otras.

El enfoque híbrido, que es el que nosotros recomendamos, consiste en determinar teóricamente la forma de las ecuaciones de recurrencia, y buscar entonces empíricamente los valores de las constantes que se utilizan en estas ecuaciones para la implementación concreta que estemos empleando. El umbral óptimo se pue-

de estimar entonces hallando el tamaño n del caso para el cual no hay diferencia entre aplicar directamente el algoritmo clásico o pasar a un nivel más de recursión; véase el problema 7.8. Por esta razón seleccionamos $n_0 = 64$: el algoritmo clásico de multiplicación requiere $h(64) = 64^2 = 4096$ microsegundos para multiplicar dos números de 64 cifras, mientras que si utilizamos un nivel de recursividad en el enfoque de divide y vencerás, esta misma multiplicación requiere $g(64) = 16 \times 64 = 1024$ microsegundos, además de tres multiplicaciones de números de 32 cifras mediante el algoritmo clásico, con un coste de $h(32) = 32^2 = 1.024$ microsegundos cada una, lo cual da el mismo total $3h(32) + g(64) = 4.096$ microsegundos.

Con esta técnica híbrida surge una dificultad práctica. Aun cuando el algoritmo clásico de multiplicación requiere un tiempo cuadrático, era una simplificación excesiva afirmar que $h(n) = cn^2$ para alguna constante c que depende de la implementación. Es más probable que existan tres constantes a, b y c tales que $h(n) = cn^2 + bn + a$. Aunque $bn + a$ resulta despreciable en comparación con cn^2 cuando n es grande, el algoritmo clásico se utiliza de hecho *precisamente* en casos de tamaño moderado. Por tanto, suele ser insuficiente estimar meramente el valor de la constante de orden más alto c. En lugar de hacer esto, es necesario medir $h(n)$ varias veces para distintos valores de n con objeto de estimar todas las constantes necesarias. El mismo comentario es aplicable a $g(n)$.

7.3 BÚSQUEDA BINARIA

La búsqueda binaria precedió a las computadoras. En esencia, es el algoritmo que se emplea para buscar una palabra en un diccionario, o un nombre en un directorio telefónico. Probablemente se trate de la aplicación más sencilla de divide y vencerás, tan sencilla que hablando con propiedad esto es una aplicación de *reducción (simplificación)* más que de divide y vencerás: la solución de todo caso suficientemente grande se reduce a un caso más pequeño, en este caso de tamaño mitad.

Sea $T[1..n]$ una matriz ordenada por orden no decreciente; esto es, $T[i] \leq T[j]$ siempre que sea $1 \leq i \leq j \leq n$. Sea x un elemento. El problema consiste en buscar x en la matriz T, si es que está. Formalmente, deseamos encontrar el índice i tal que $1 \leq i \leq n + 1$ y $T[i - 1] < x \leq T[i]$, con la convención lógica consistente en que $T[0] = -\infty$ y $T[n + 1] = +\infty$ (al decir *convenio lógico* queremos indicar que estos valores no están realmente presentes en la matriz). La aproximación evidente para este problema consiste en examinar secuencialmente todos los elementos de T, hasta que o bien lleguemos al final de la matriz, o bien encontremos un elemento que no sea menor que x:

```
función secuencial(T[1..n], x)
    {Búsqueda secuencial de x en una matriz }
    para i ← 1 hasta n hacer
        si T[i] ≥ x entonces devolver i
    devolver n + 1
```

Este algoritmo requiere claramente un tiempo que está en $\Theta(r)$, en donde r es el índice que se devuelve. Esto está en $\Omega(n)$ en el caso peor, y $O(1)$ en el caso mejor. Si suponemos que los elementos de T son diferentes, que x está realmente en algún lugar de la matriz, y que tiene la misma probabilidad de encontrarse en cualquier posición, entonces el número medio de pasadas por el bucle es $(n + 1)/2$; véase el Problema 7.9. Así pues, tanto en el caso promedio como en el caso peor, la búsqueda secuencial requiere un tiempo que está en $\Theta(n)$.

Para acelerar la búsqueda, deberíamos buscar x o bien en la primera mitad de la matriz, o bien en la segunda. Para averiguar cuál de estas búsquedas es la correcta, comparamos x con un elemento de la matriz. Sea $k = \lceil n / 2 \rceil$. Si $x \leq T[k]$, entonces se puede restringir la búsqueda de x a $T[1..k]$; en caso contrario basta con buscar en $T[k+1..n]$. Para evitar comparaciones repetidas en cada llamada recursiva, es mejor verificar desde un comienzo si la respuesta es $n + 1$, esto es, *si x está a la derecha de T*. Obtenemos el algoritmo siguiente, que se ilustra en la figura 7.1.

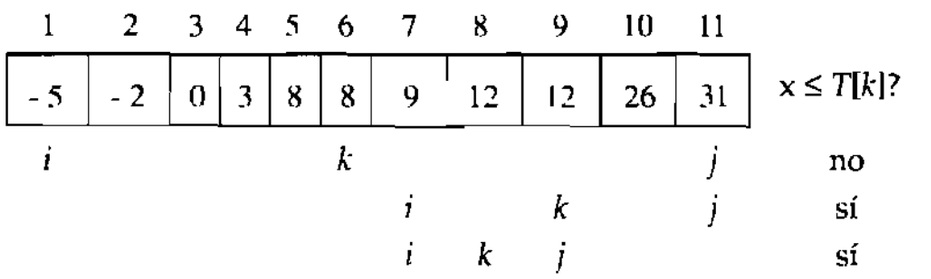


Figura 7.1. Búsqueda binaria de x = 12 en T[1..11]

```
función busquedabin(T[1..n], x)
    si n = 0 o x > T[n] entonces devolver n + 1
    sino devolver binrec(T[1..n], x)

función binrec(T[i..j],x)
    {Búsqueda binaria de x en la submatriz T[i..j]
    con la seguridad de que T[i − 1] < x ≤ T[j] }
    si i = j entonces devolver i
    k ← (i + j) ÷ 2
    si x ≤ T[k] entonces devolver binrec(T[i..k], x)
                sino devolver binrec(T[k + 1..j], x)
```

Sea $t(m)$ el tiempo requerido por una llamada a *binrec*$(T[i..j], x)$, en donde $m = j - i + 1$ es el número de elementos que restan a efectos de búsqueda. El tiempo requerido por una llamada a *busquedabin*$(T[i..j], x)$ es claramente $t(n)$ salvo por una pequeña constante aditiva.

Cuando $m > 1$, el algoritmo requiere una cantidad constante de tiempo además de una llamada recursiva con $\lceil m / 2 \rceil$ o $\lfloor m / 2 \rfloor$ elementos, dependiendo de si

$x \leq T[k]$ o no. Por tanto, $t(n) = t(n/2) + g(n)$ cuando n es par, y donde $g(n) \in O(1) = O(n^0)$. Por nuestro análisis general de los algoritmos de divide y vencerás, utilizando la ecuación 7.1 con $a = 1, b = 2$ y $k = 0$, concluimos que $t(n) \in \Theta(\log n)$. Por tanto, la búsqueda binaria se puede efectuar en un tiempo logarítmico en el caso peor. Resulta fácil ver que esta versión de la búsqueda binaria también requiere un tiempo logarítmico aun en el caso mejor.

Dado que la llamada recursiva *está dinámicamente* al final del logaritmo, es fácil realizar una versión iterativa.

```
función biniter(T[1..n], x)
    {búsqueda binaria iterativa de x en la matriz T}
    si x > T[n] entonces devolver n + 1
    i ← 1; j ← n
    mientras i < j hacer
        {T[i − 1] < x ≤ T[j]}
        k ← (i + j) ÷ 2
        si x ≤ T[k] entonces j ← k
                       sino i ← k + 1
    devolver i
```

El análisis de este algoritmo es idéntico al de su contrapartida recursiva, *busquedabin*. Se comprueban exactamente las mismas posiciones de la matriz (salvo cuando $n = 0$; véase el Problema 7.10), y los índices i, j y k asumen exactamente la misma secuencia de valores. Por tanto, la búsqueda binaria iterativa también requiere un tiempo logarítmico, tanto en el caso peor como en el mejor. Este algoritmo se puede modificar para hacerlo más rápido en el caso mejor (tiempo constante) a costa de hacerlo ligeramente más lento (aunque sigue siendo logarítmico) en el caso peor, pero esto va en detrimento del comportamiento en el caso intermedio para casos muy grandes; véase el Problema 7.11.

7.4 ORDENACIÓN

Sea $T[1..n]$ una matriz de n elementos. Nuestro problema es ordenar estos elementos por orden ascendente. Ya hemos visto que el problema se puede resolver mediante *ordenación por selección* y *ordenación por inserción* (Sección 2.4), o bien mediante *ordenación por montículo (heapsort)* (Sección 5.7). Recordemos que el análisis en los casos peor y promedio muestra que este último método requiere un tiempo que está en $\Theta(n \log n)$, mientras que los dos métodos anteriores requieren un tiempo cuadrático. Hay varios algoritmos clásicos de ordenación que siguen el esquema de divide y vencerás. Es interesante observar lo diferentes que son: aun habiendo decidido resolver un problema por divide y vencerás, disponemos de amplio margen de maniobra para nuestra creatividad. Estudiaremos ahora dos de ellos —*ordenar por fusión (mergesort)* y *ordenación rápida (quicksort)*— y dejaremos otro más para el Capítulo 11.

7.4.1 Ordenación por fusión

El enfoque evidente de divide y vencerás para este problema consiste en descomponer la matriz T en dos partes cuyos tamaños sean tan parecidos como sea posible, ordenar estas partes mediante llamadas recursivas, y después fusionar las soluciones de cada parte, teniendo buen cuidado en mantener el orden. Para hacer esto, necesitamos un algoritmo eficiente para fusionar dos matrices ordenadas U y V en una única matriz T cuya longitud sea la suma de las longitudes de U y V. Esto se puede lograr de forma más eficiente —y más sencilla— si se dispone de espacio adicional al final de las matrices U y V, para utilizarlo como centinela. (Esta técnica sólo funciona si podemos dar al centinela un valor previamente acordado y que estemos seguros que es mayor que cualquier elemento de U y V, valor que representaremos por «∞»; véase el problema 7.13):

```
procedimiento fusionar(U[1..m+1], V[1..n+1], T[1..m+n])
    {Fusiona las matrices ordenadas U[1..m] y V[1..n]
    almacenándolas en T[1..m+n], U[m+1] y V[n+1]
    se utilizan como centinelas }
    i, j ← 1
    U[m+1], V[n+1] ← ∞
    para k ← 1 hasta m + n hacer
        si U[i] < V[j]
        entonces T[k] ← U[i]; i ← i + 1
        sino T[k] ← V[j]; j ← j + 1
```

El algoritmo de ordenación por fusión es como sigue, en donde utilizamos la ordenación por inserción (*insertar*) de la Sección 2.4 como subalgoritmo básico. Para mejorar la eficiencia, puede ser mejor que las matrices intermedias U y V sean variables globales:

```
procedimiento ordenarporfusión(T[1..n])
    si n es suficientemente pequeño entonces insertar (T)
    sino
        matriz U[1..1 + ⌊n / 2⌋], V[1..1 + ⌊n / 2⌋],
        U[1..⌊n / 2⌋] ← T[1..⌊n / 2⌋]
        V[1..⌈n / 2⌉] ← T[1 + ⌊n / 2⌋..n]
        ordenarporfusión(U[1..⌊n / 2⌋]
        ordenarporfusión(V[1..⌈n / 2⌉]
        fusionar(U, V, T)
```

La figura 7.2 muestra la forma en que funciona la ordenación por fusión.

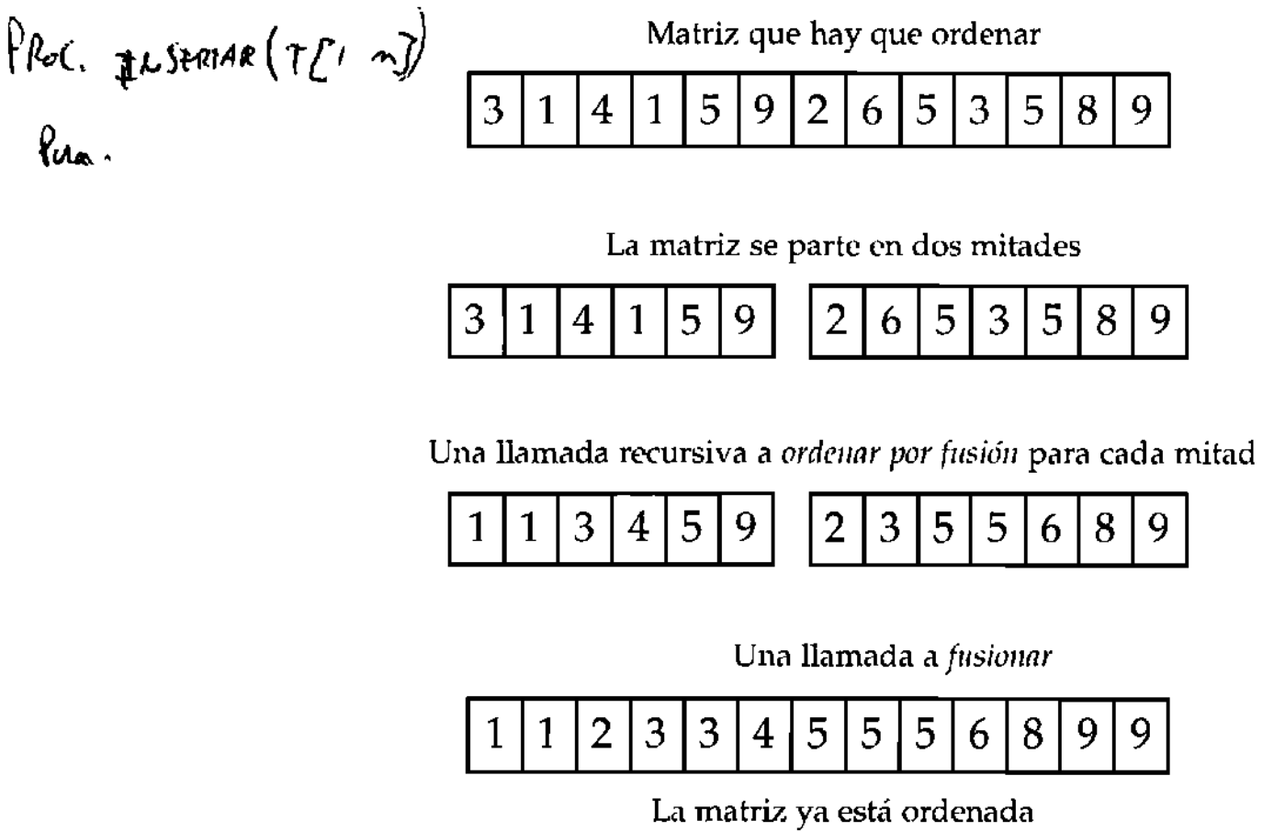


Figura 7.2. Ordenar por fusión

Este algoritmo de ordenación ilustra bien todas las características de divide y vencerás. Cuando el número de elementos que hay que ordenar es pequeño, se utiliza un algoritmo relativamente sencillo. Por otra parte, cuando está justificado por el número de elementos, *ordenarporfusión* separa el ejemplar en dos subejemplares de tamaño mitad, resuelve los dos recursivamente, y entonces combina las dos medias matrices ya ordenadas para obtener la solución del ejemplar original.

Sea $t(n)$ el tiempo requerido por este algoritmo para ordenar una matriz de n elementos. La separación de T en U y V requiere un tiempo lineal. Es fácil ver que *fusionar*(U, V, T) también requiere un tiempo lineal Consiguientemente, $t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + g(n)$, en donde $g(n) \in \Theta(n)$. Esta recurrencia, que pasa a ser $t(n) = 2t(n/2) + g(n)$ cuando n es par, es un caso especial de nuestro análisis general para algoritmos de divide y vencerás. La Ecuación 7.1 es aplicable con $a = 2$, $b = 2$ y $k = 1$. Dado que $a = b^k$, se aplica el segundo caso para dar $t(n) \in \Theta(n \log n)$. Por tanto, la eficiencia de *ordenar por fusión* es similar a la de *ordenación por montículo*. La ordenación por fusión puede ser ligeramente más rápida en la práctica, pero requiere una cantidad significativamente mayor de espacio para las matrices intermedias U Y V. Recuerde que *ordenación por montículo* puede ordenar *in situ*, en el sentido de que solamente necesita un pequeño número constante de variables de trabajo. La ordenación por fusión también se puede implementar *in situ*, pero a costa de tal incremento de la constante oculta que esto sólo tiene un interés teórico; véase el problema 7.14.

El algoritmo de ordenación por fusión ilustra la importancia de crear subejemplares de tamaño aproximadamente igual al desarrollar algoritmos de divide y vence-

rás. Considérese la siguiente variante de *ordenarporfusión*. (La llamada muda —ficticia— a *ordenarporfusiónmala*(V[1..1]) sólo se incluye para acentuar la similitud con el algoritmo *ordenarporfusión* original.)

```
procedimiento ordenarporfusiónmala(T[1..n])
    si n es suficientemente pequeño entonces insert(T)
    sino
        matriz U[1..n], V[1..2]
        U[1..n-1] ← T[1..n-1]
        V[1] ← T[n]
        ordenarporfusiónmala(U[1..n-1])
        ordenarporfusiónmala(V[1..1])
        fusionar(U, V, T)
```

Sea $\hat{t}(n)$ el tiempo necesario para ordenar n elementos con este algoritmo modificado. Está claro que $\hat{t}(n) = \hat{t}(n-1) + \hat{t}(1) + \hat{g}(n)$, donde $\hat{g}(n) \in \Theta(n)$. Esta recurrencia da lugar a $\hat{t}(n) \in \Theta(n^2)$. Por tanto, olvidarnos de equilibrar los tamaños de los subcasos puede ser desastroso para la eficiencia de un algoritmo obtenido por divide y vencerás. De hecho, *ordenarporfusiónmala* no es más que ¡una implementación poco eficiente de la ordenación por inserción!

7.4.2 Ordenación rápida (Quicksort)

El algoritmo de ordenación inventado por Hoare, que se suele conocer con el nombre de *quicksort* u ordenación rápida, también esta basado en el principio de divide y vencerás. A diferencia de *ordenar por fusión*, la mayor parte del trabajo no recursivo que hay que hacer se invierte en construir los subcasos, y no en combinar sus soluciones. Como primer paso, el algoritmo selecciona como *pivote* uno de los elementos de la matriz que haya que ordenar. A continuación, la matriz se parte a ambos lados del pivote: se desplazan los elementos de tal manera que los que sean mayores que el pivote queden a su derecha, mientras que los demás quedan a su izquierda. Si ahora las partes de la matriz que quedan a ambos lados del pivote se ordenan independientemente mediante llamadas recursivas al algoritmo, el resultado final es una matriz completamente ordenada; no hace falta un paso subsiguiente de fusión. Para equilibrar los tamaños de los dos subcasos que hay que ordenar, nos gustaría utilizar el elemento *mediana* como pivote (para una definición de la mediana, véase la Sección 7.5). Desafortunadamente, encontrar de la mediana requiere un tiempo excesivo. Por esta razón, nos limitamos a utilizar como pivote un elemento arbitrario de la matriz, y esperamos tener suerte.

El diseño de un algoritmo de partición con tiempo lineal no es tarea difícil. Sin embargo, en la práctica resulta crucial que la constante oculta sea pequeña, para que *quicksort* sea competitivo con otras técnicas de ordenación tales como *ordenación por montículo*. Supongamos que es preciso descomponer la submatriz $T[i..j]$ empleando como pivote $p = T[i]$. Una buena forma de hacer la descomposición consiste en explorar la submatriz una sola vez, pero empezando por los dos extremos. Los punte-

ros k y l se inicializan i y j+1 respectivamente. A continuación, se incrementa el puntero k hasta que $T[k] > p$, y se decrementa el puntero l hasta que $T[l] \leq p$. Ahora se intercambian $T[k]$ y $T[l]$. Este proceso continúa mientras sea $k < l$. Finalmente, se intercambian $T[i]$ y $T[l]$ para poner el pivote en posición correcta:

```
procedimiento pivote(T[i..j]; var l)
    {Permuta los elementos de la matriz T[i..j] y proporciona un valor l
    tal que, al final, i ≤ l ≤ j; T[k] ≤ p para todo i ≤ k < l, T[l] = p,
    y T[k] > p para todo l < k ≤ j, en donde p es el valor inicial de T[i]}
    p ← T[i]
    k ← i; l ← j+1
    repetir k ← k+1 hasta que T[k] > p o k ≥ j
    repetir l ← l−1 hasta que T[l] ≤ p
    mientras k < l hacer
        intercambiar T[k] y T[l]
        repetir k ← k+1 hasta que T[k] > p
        repetir l ← l−1 hasta que T[l] ≤ p
    intercambiar T[i] y T[l]
```

Y lo que sigue es el algoritmo de ordenación. Para ordenar la matriz completa, basta llamar a *quicksort*(T[1..n]):

```
procedimiento quicksort(T[i..j])
    {Ordena la submatriz T[i..j] por orden no decreciente}
    si j − i es suficientemente pequeño entonces insertar (T[i..j])
    sino
        pivote(T[i..j], l)
        quicksort(T[i..l−1])
        quicksort(T[l+1..j]
```

La figura 7.3 muestra cómo funcionan *pivote* y *quicksort*.

Quicksort es ineficiente si sucede que de forma sistemática, en la mayoría de las llamadas recursivas, los subejemplares $T[i..l-1]$ y $T[l+1..j]$ están fuertemente desequilibrados. En el caso peor, por ejemplo si T ya está ordenado antes de la llamada a *quicksort*, obtenemos $l = i$ en todas las ocasiones, lo cual implica una llamada recursiva a un caso de tamaño cero y otra a un caso cuyo tamaño sólo se reduce en una unidad. Esto da lugar a una recurrencia similar a la que encontrábamos en el análisis de *ordenarporfusiónmala*, la versión no equilibrada de *ordenarporfusión*. Una vez más, el tiempo de ejecución es cuadrático. Por tanto, *quicksort* requiere en el caso peor un tiempo que está en $\Omega(n^2)$ para ordenar n elementos.

Por otra parte, si la matriz que hay que ordenar se encuentra inicialmente en orden aleatorio, entonces es probable que la mayoría de los subejemplares que haya que ordenar estén suficientemente bien equilibrados. Para determinar el tiempo *promedio* que requiere *quicksort* para ordenar una matriz de n elementos, es preciso hacer una

suposición acerca de la distribución de probabilidades de los casos de n elementos. La suposición más natural es que los elementos de T son diferentes, y que todas las $n!$ permutaciones iniciales posibles de los elementos son igualmente probables. Hay que poner de manifiesto, sin embargo, que esa suposición puede no ser válida (incluso puede ser totalmente equivocada) para algunas aplicaciones, en cuyo caso el análisis siguiente no sería aplicable. Tal cosa ocurre, por ejemplo, si nuestra aplicación necesita ordenar matrices que ya estén casi totalmente ordenadas.

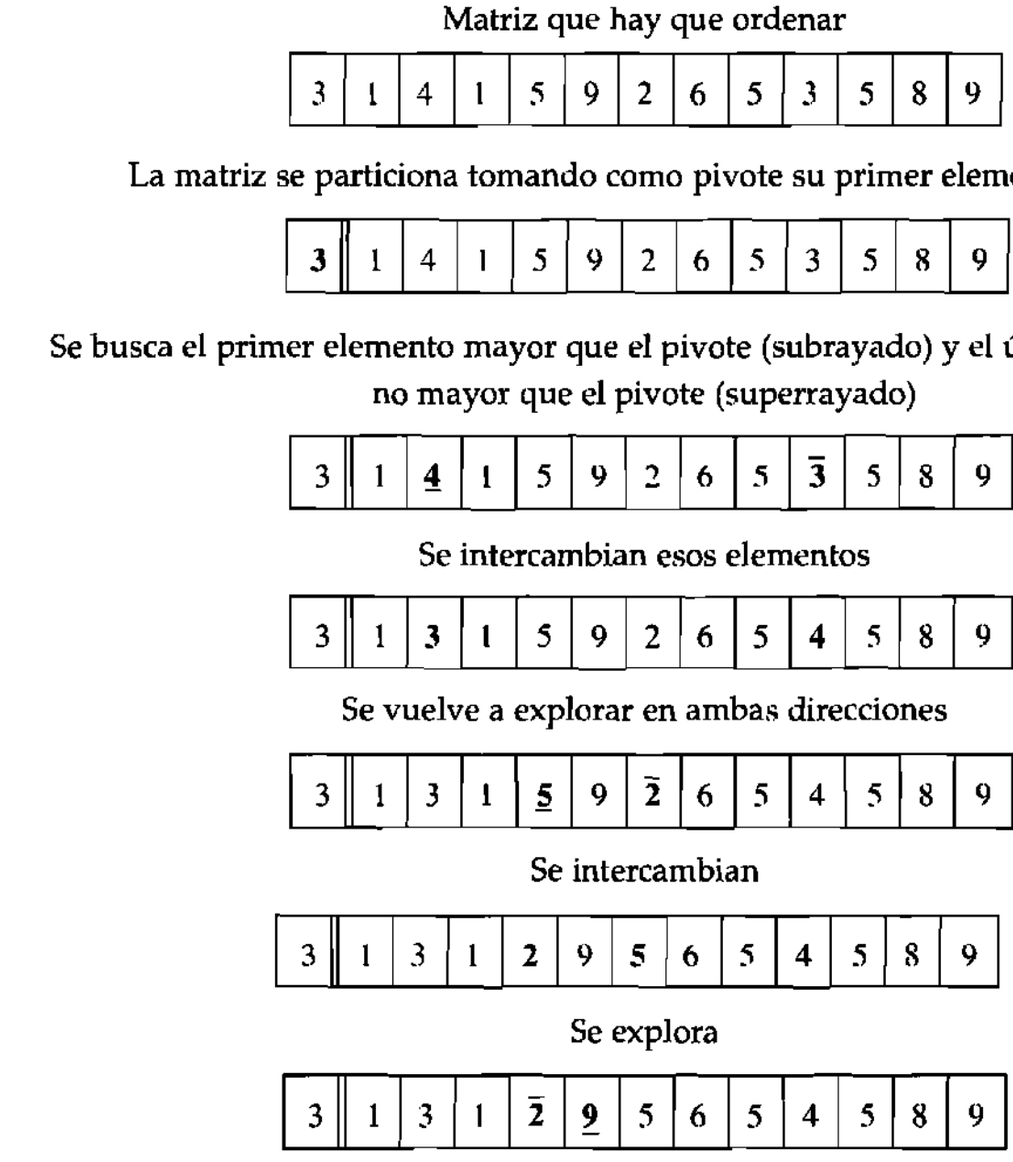


Figura 7.3. quicksort (ordenación rápida)

Sea $t(m)$ el tiempo medio que requiere una llamada a *quicksort*($T[i..j]$), donde $m = j - i + 1$ es el número de elementos que hay en la submatriz. En particular, *quicksort*($T[1..n]$) requiere un tiempo $t(n)$ para ordenar los n elementos de la matriz. Por nuestra suposición acerca de la distribución de probabilidad de los casos, el pivote seleccionado por el algoritmo cuando se le pide ordenar $T[1..n]$ se encuentra con igual probabilidad en cualquier posición con respecto a los demás elementos de T. Por tanto, el valor de l que devuelve el algoritmo de partición después de la llamada inicial *pivote*($T[1..n]$, l) puede ser cualquier entero entre 1 y n, teniendo todos los valores la misma probabilidad $1/n$. Esta operación de pivote requiere un tiempo lineal $g(n) \in \Theta(n)$. Quedan por ordenar recursivamente dos submatrices de tamaños $l - 1$ y $n - l$ respectivamente. Se puede demostrar que la distribución de probabilidad de las submatrices sigue siendo uniforme; véase el Problema 7.16. Por tanto, el tiempo medio requerido para ejecutar estas llamadas recursivas es $t(l - 1) + t(n - l)$. Como consecuencia:

$$t(n) = \frac{1}{n}\sum_{l=1}^{n}\left(g(n) + t(l-1) + t(n-l)\right)$$

siempre que n sea suficientemente grande para que merezca la pena emplear el enfoque recursivo. En esta fórmula, $\frac{1}{n}$ es la probabilidad de que cualquier valor dado de l entre 1 sea devuelto por la llamada de mayor nivel a *pivote*, y $g(n) + t(l - 1) + t(n - l)$ es el tiempo esperado para ordenar n elementos con la condición de que este valor de l sea devuelto por esa llamada.

Para hacer más explícita la fórmula, sea n_0 el umbral por encima del que se utiliza el enfoque recursivo, lo cual quiere decir que se utiliza la ordenación por inserción siempre que no haya más de n_0 elementos para ordenar. Además, sea d una constante (que dependerá de la implementación) tal que $g(n) \leq dn$ siempre que $n > n_0$. Si sacamos $g(n)$ del sumatorio, tenemos

$$t(n) \leq dn + \frac{2}{n}\sum_{l=1}^{n}\left(t(l-1) + t(n-l)\right) \quad \text{para } n > n_0$$

Si observamos que el término $t(k)$ aparece dos veces por cada $0 \leq k \leq n - 1$, una sencilla manipulación da lugar a

$$t(n) \leq dn + \frac{2}{n}\sum_{k=0}^{n-1} t(k) \text{ para } n > n_0 \tag{7.2}$$

Una ecuación de este tipo es más difícil de analizar que las recurrencias lineales que vimos en la Sección 4.7. En particular, la ecuación 7.1 no puede aplicarse en esta ocasión. Por analogía con *ordenar por fusión*, resulta sin embargo razonable esperar que $t(n)$ esté en $O(n \log n)$: en término medio, los subejemplares no están demasiado desequilibrados y la solución estaría en $O(n \log n)$ si estuvieran tan bien equilibra-

dos como con *ordenar por fusión*. La demostración de esta conjetura es una bonita aplicación de la técnica de inducción constructiva (Sección 1.6.4). Para aplicar esta técnica, postulamos la existencia de una constante c, desconocida por el momento, tal que $t(n) \leq c\, n \log n$ para todo $n \geq 2$. Hallaremos un valor adecuado para esta constante en el proceso de demostración de su existencia por inducción matemática generalizada. Comenzamos con $n = 2$, porque $n \log n$ está indefinido o es nulo para valores más pequeños de n; alternativamente, podríamos empezar en $n = n_0 + 1$.

> Teorema 7.4.1 *Quicksort requiere un tiempo en $O(n \log n)$ para ordenar n elementos en el caso medio.*

Demostración. Sea $t(n)$ el tiempo requerido por *quicksort* para ordenar n elementos en el caso promedio. Sean d y n_0 constantes tales que es válida la ecuación 7.2. Deseamos demostrar que $t(n) \leq c\, n \log n$ para todo $n \geq 2$, siempre y cuando c sea una constante elegida adecuadamente. Procedemos por inducción constructiva. Supongamos sin pérdida de generalidad que $n_0 \geq 2$.

◊ *Base*: consideremos cualquier entero n tal que $2 \leq n \leq n_0$. Tenemos que demostrar que $t(n) \leq c\, n \log n$. Esto es fácil, puesto que tenemos total libertad para seleccionar la constante c, y el número de casos es finito. Basta con seleccionar c que sea al menos tan grande como $t(n)/(n \log n)$. De esta manera, nuestra primera restricción sobre c es:

$$c \geq \frac{t(n)}{n \log n} \text{ para todo } n \text{ tal que } 2 \leq n \leq n_0 \tag{7.3}$$

◊ *Paso de inducción*: consideremos cualquier entero $n > n_0$. Se toma como hipótesis de inducción que $t(k) \leq c\, k \log k$ para todo k tal que $2 \leq k < n$. Deseamos restringir c de tal manera que $t(n) \leq c\, n \log n$ se siga de la hipótesis de inducción. Supongamos que a representa $t(0) + t(1)$. Comenzando con la ecuación 7.2:

$$\begin{aligned} t(n) &\leq dn + \frac{2}{n}\sum_{k=0}^{n-1} t(k) \\ &= dn + \frac{2}{n}\left(t(0) + t(1) + \sum_{k=2}^{n-1} t(k) \right) \\ &\leq dn + \frac{2a}{n} + \frac{2}{n}\sum_{k=2}^{n-1} ck \log k \text{ por la hipótesis de inducción} \\ &\leq dn + \frac{2a}{n} + \frac{2c}{n}\int_2^n x \log x\, dx \text{ (véase la figura 7.4)} \end{aligned}$$

$$= dn + \frac{2a}{n} + \frac{2c}{n}\left[\frac{x^2 \log x}{2} - \frac{x^2}{4}\right]_{x=2}^{n}$$

(recordemos que «log» representa el logaritmo natural)

$$< dn + \frac{2a}{n} + \frac{2c}{n}\left(\frac{n^2 \log n}{2} - \frac{n^2}{4}\right)$$

$$= dn + \frac{2a}{n} cn \log n - \frac{cn}{2}$$

$$= cn \log n - \left(\frac{c}{2} - d - \frac{2a}{n^2}\right) n.$$

Se sigue que $t(n) \leq c\, n \log n$ siempre y cuando $c/2 - d - 2a/n^2$ sea no negativo, lo cual equivale a decir que $c \geq 2d + 4a/n^2$. Dado que aquí consideramos solamente el caso en el cual $n > n_0$, todo va bien siempre y cuando

$$c \geq 2d + \frac{4a}{(n_0 + 1)^2} \tag{7.4}$$

que es nuestra segunda y última restricción sobre c.
Reuniendo las restricciones dadas por las ecuaciones 7.3 y 7.4, basta hacer

$$c = \text{máx}\left(2d + \frac{4(t(0) + t(1))}{(n_0 + 1)^2}, \text{máx}\left\{\frac{t(n)}{n \log n} \middle| 2 \leq n \leq n_0\right\}\right) \tag{7.5}$$

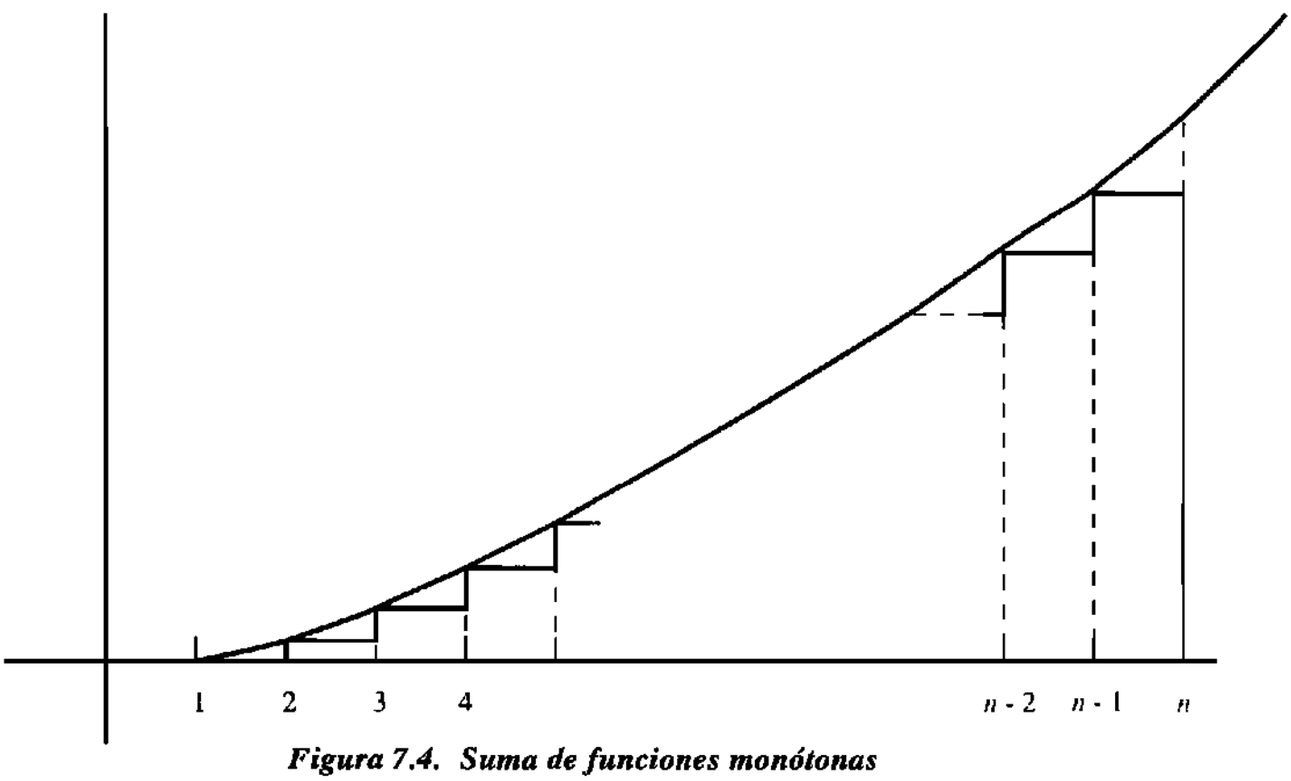


Figura 7.4. Suma de funciones monótonas

para finalizar la demostración por inducción constructiva de que $t(n) \leq c\, n \log n$ para todo $n \geq 2$, y por tanto que $t(n) \in O(n \log n)$.
Si se siente confuso o poco convencido, le animamos a que desarrolle por sí mismo una demostración por inducción matemática generalizada —en vez de inducción matemática constructiva— de que $t(n) \leq c\, n \log n$ para todo $n \geq 2$, utilizando esta vez el valor explícito de c dado por la ecuación 7.5.

Por tanto, *quicksort* puede ordenar una matriz de n elementos distintos en un tiempo promedio que está en $O(n \log n)$. En la práctica, la constante oculta es más pequeña que las asociadas en *ordenación por montículo* o en *ordenar por fusión*. Si ocasionalmente se puede tolerar algún tiempo de ejecución largo, éste es un excelente algoritmo de propósito general. ¿Se puede modificar *quicksort* para que requiera un tiempo que esté en $O(n \log n)$ incluso en el caso peor? La respuesta es sí pero no. Incluso si seleccionamos la mediana de $T[i..j]$ como pivote, lo cual se puede hacer en un tiempo lineal según se verá en la Sección 7.5, *quicksort* tal como se ha descrito aquí sigue requiriendo un tiempo cuadrático en el caso peor, lo cual sucede si son iguales todos los elementos que hay que ordenar. Basta una sencilla modificación del algoritmo de partición para evitar este comportamiento incorrecto, aunque es complicado programarlo de forma correcta y eficiente. Para esto, se necesita un nuevo

procedimiento *pivotebis*($T[i..j]$, p; **var** k, l)

que descomponga T en *tres* secciones, empleando p como pivote: después de la partición, los elementos de $T[i..k]$ son más pequeños que p, los de $T[k+1..l-1]$ son iguales a p, y los de $T[l..j]$ son más grandes que p. Los valores de k y l son devueltos por *pivotebis*. Después de hacer la partición con una llamada a *pivotebis*($T[i..j]$, $T[i]$, k, l) hay que llamar a *quicksort* recursivamente, con $T[i..k]$ y $T[l..j]$. Con esta modificación, la ordenación de una matriz de elementos iguales requiere un tiempo lineal. Hay algo más interesante: ahora *quicksort* requiere un tiempo que está en $O(n \log n)$ incluso en el caso peor si se selecciona como pivote la mediana de $T[i..j]$ en un tiempo lineal. Sin embargo, sólo mencionamos esta posibilidad para indicar que hay que evitarla: la constante oculta asociada a esta versión «mejorada» de *quicksort* es tan grande que da lugar a un algoritmo, peor que *ordenación por montículo* en todos los casos.

7.5 BÚSQUEDA DE LA MEDIANA

Sea $T[1..n]$ una matriz de enteros, y sea s un entero entre 1 y n. Se define el *s-ésimo menor elemento* de T como aquel elemento que se encontraría en la s-ésima posición si se ordenara T en orden no decreciente. Dados T y s, el problema de encontrar el s-ésimo elemento de T se conoce como el *problema de selección*. En particular, se define la *mediana* de T como su $\lceil n/2 \rceil$-ésimo elemento. Cuando n es

impar y los elementos de T son diferentes, la mediana es simplemente aquel elemento de T tal que en T hay tantos elementos más pequeños que él como elementos mayores que él. Por ejemplo, la mediana de [3, 1, 4, 1, 5, 9, 2, 6, 5] es 4, puesto que 3, 1, 1 y 2 son más pequeños que 4, mientras que 5, 9, 6 y 5 son mayores.

¿Qué podría ser más sencillo que buscar el menor elemento de T, o calcular la media de todos los elementos? Sin embargo, no es evidente que sea posible calcular la mediana con tanta facilidad. Un algoritmo sencillo para determinar la mediana de $T[1..n]$ consiste en ordenar la matriz y extraer entonces el elemento $\lceil n/2 \rceil$*–ésimo*. Si utilizamos *ordenación por montículo* u *ordenación por fusión*, esto requiere un tiempo que está en $\Theta(n \log n)$. ¿Podemos hacerlo mejor? Para responder a esta pregunta, estudiaremos la interrelación entre hallar la mediana y seleccionar el s-ésimo elemento más pequeño.

Es evidente que todo algoritmo para el problema de selección se puede utilizar para hallar la mediana: basta con seleccionar el $\lceil n / 2 \rceil$-ésimo elemento más pequeño. Curiosamente, la inversa también es cierta. Supongamos por el momento que está disponible un algoritmo *mediano*($T[1..n]$) que devuelve la mediana de T. Dada una matriz T y un entero s, ¿cómo se podría utilizar este algoritmo para determinar el s-ésimo elemento más pequeño de T? Sea p la mediana de T. Ahora usamos p como pivote, de forma muy parecida a *quicksort*, pero usando el algoritmo *pivotebis* presentado al final de la sección anterior. Recuerde que una llamada a *pivotebis*($T[1..n]$, p; **var** k, l) particiona a $T[i..j]$ en tres secciones: T se reorganiza de tal manera que los elementos de $T[i..k]$ sean más pequeños que p, los de $T[k+1..l-1]$ sean iguales a p, y los de $T[l..j]$ sean mayores que p. Tras una llamada a *pivotebis*(T, p, k, l), hemos terminado si $k < s < l$, puesto que entonces el s-ésimo elemento más pequeño de T es igual a p. Si $s \leq k$, entonces el s-ésimo elemento más pequeño de T es ahora el s-ésimo elemento más pequeño de $T[1..k]$. Por último, si $s \geq l$, entonces el s-ésimo elemento más pequeño de T es ahora el $(s - l + 1)$-ésimo elemento más pequeño de $T[l..n]$. En cualquier caso, hemos avanzado, porque o bien hemos terminado, o bien la submatriz que hay que considerar contiene menos de la mitad de los elementos, por cuanto p es la mediana de la matriz original.

Hay una gran similitud entre este enfoque y la búsqueda binaria (Sección 7.3), y de hecho el algoritmo resultante se puede programar de forma iterativa en lugar de hacerlo recursivamente. La idea clave consiste en emplear dos variables i y j, inicializadas a 1 y n respectivamente, y asegurarnos que en todo momento $i \leq s \leq j$, y que los elementos de $T[1..i-1]$ sean más pequeños que los de $T[i..j]$, que a su vez son más pequeños que los de $T[j+1..n]$. La consecuencia inmediata es que el elemento deseado se encuentra en $T[i..j]$. Cuando todos los elementos de $T[i..j]$ sean iguales, hemos acabado.

La figura 7.5 ilustra el proceso. Por sencillez, la ilustración supone que *pivotebis* se ha implementado de una manera que es intuitivamente sencilla, aun cuando una implementación realmente eficiente haría las cosas de otra manera.

```
función selección(T[1..n], s)
    {Busca el s-ésimo elemento más pequeño de T, 1 ≤ s ≤ n}
    i ← 1; j ← n
    repetir
        {La respuesta se encuentra en T[i..j]}
        p ← mediana(T[i..j])
        pivotebis(T[i..j], p, k, l)
        si s ≤ k entonces j ← k
        sino si s ≥ l entonces i ← l
             sino devolver p
```

Matriz en la que hay que hallar el 4.° elemento más pequeño

3	1	4	1	5	9	2	6	5	3	5	8	9

Se particiona la matriz usando como pivote $p = 5$, con *pivotebis*

3	1	4	1	2	3	5	5	5	9	6	8	9

Sólo es relevante la parte a la izquierda del pivote, puesto que $4 \leq 7$

3	1	4	1	2	3	•	•	•	•	•	•	•

Se particiona esa parte usando como pivote su mediana $p = 2$

1	1	2	3	4	3	•	•	•	•	•	•	•

Sólo es relevante la parte a la derecha del pivote puesto que $4 \geq 4$

•	•	•	3	4	3	•	•	•	•	•	•	•

Se particiona esa parte usando como pivote su mediana $p = 3$

•	•	•	3	3	4	•	•	•	•	•	•	•

La respuesta es 3, porque el pivote está en la 4.ª posición

Figura 7.5. Selección empleando la mediana

Mediante un análisis similar al de la búsqueda binaria, el algoritmo anterior selecciona el elemento requerido de T pasando por el bucle **repetir** un número logarítmico de veces en el caso peor. Sin embargo, las pasadas por el bucle ya no requieren un tiempo constante y lo cierto es que este algoritmo no se podrá utilizar

mientras no dispongamos de una forma eficiente de hallar la mediana, que era nuestro problema original. ¿Podemos modificar el algoritmo para evitar recurrir a la mediana?

En primer lugar, observemos que nuestro algoritmo sigue funcionando independientemente del elemento de T que seleccionemos como pivote (el valor de p). Solamente la *eficiencia* del algoritmo depende de la elección de pivote: el uso de la mediana asegura que el número de elementos que estamos considerando se divide al menos por dos en cada pasada por el bucle **repetir**. Si estamos dispuestos a sacrificar la velocidad en el caso peor para obtener un algoritmo razonablemente rápido de análisis en el caso promedio, podemos tomar prestada otra idea de *quicksort*, y seleccionar simplemente $T[i]$ como pivote. En otras palabras, sustituimos la primera instrucción del bucle por

$$p \leftarrow T[i]$$

Esto da lugar a que el algoritmo invierta un tiempo cuadrático en el caso peor, por ejemplo si la matriz está en orden decreciente y deseamos encontrar el elemento más pequeño. Sin embargo, este algoritmo modificado funciona en un tiempo *lineal* de análisis en el caso promedio, con la suposición habitual de que los elementos de T sean distintos y que las $n!$ permutaciones de los elementos iniciales posibles sean igualmente probables. (El análisis es paralelo al de *quicksort*, véase el problema 7.18.) Esto es mucho mejor que el tiempo que se necesita de análisis en el caso promedio si realmente ordenamos la matriz, pero el comportamiento en el caso peor no es admisible para muchas aplicaciones.

Felizmente, este caso peor cuadrático se puede evitar sin sacrificar el comportamiento lineal de análisis en el caso promedio. La idea es que el número de pasadas por el bucle sigue siendo logarítmica siempre y cuando seleccionemos un pivote razonablemente próximo a la mediana. Una buena *aproximación* a la mediana se puede hallar rápidamente con un poco de astucia. Consideremos el algoritmo siguiente:

```
función pseudomediana(T[1..n])
    {Busca una aproximación a la mediana de la matriz T}
    si n ≤ 5 entonces devolver medianadhoc(T)
    z ← ⌊n/5⌋
    matriz Z[1..z]
    para i ← 1 hasta z hacer Z[i] ← medianadhoc(T[5i − 4..5i])
    devolver selección(Z, ⌈z/2⌉)
```

Aquí, *medianadhoc* es un algoritmo diseñado especialmente para hallar la mediana de un máximo de 5 elementos, lo cual se puede hacer en un tiempo limitado superiormente por una constante, y *selección*$(Z, \lceil z/2 \rceil)$ determina la mediana exacta de la matriz Z. Sea p el valor devuelto por una llamada a *pseudomediana*(T). ¿A qué distancia de la verdadera mediana de T puede estar p cuando $n > 5$?

Tal como en el algoritmo, sea z igual a $\lfloor n / 5 \rfloor$, el número de elementos de la matriz Z creada por la llamada a *pseudomediana*(T). Para todo i entre 1 y z, $Z[i]$ es por definición la mediana de $T[5i - 4..5i]$ y por tanto al menos tres elementos de los cinco que hay en la matriz son menores o iguales que él. Además, como p es la verdadera mediana de Z, al menos $z/2$ elementos de Z son menores o iguales que p. Por transitividad, ($T[j] \leq Z[i] \leq p$ implica que $T[j] \leq p$), al menos $3z/2$ elementos de T son menores o iguales que p. Dado que es $z = \lfloor n / 5 \rfloor \geq (n - 4) / 5$, concluimos que al menos $(3n - 12)/10$ elementos de T son menores o iguales que p, y por tanto como mucho los $(7n+12)/10$ elementos restantes de T son estrictamente mayores que p. El mismo razonamiento se aplica al número de elementos de T que son estrictamente *más pequeños* que p.

Aun cuando es probable que p no sea la mediana exacta de T, concluimos que su rango está aproximadamente entre $3n/10$ y $7n/10$. Para visualizar la forma en que surgen estos factores, aunque no hay nada en la ejecución de *pseudomediana* que realmente se corresponda con esta ilustración, consideremos cómo en la figura 7.6 que los elementos de T están dispuestos en cinco filas, con la posible excepción de un máximo de 4 elementos, que apartamos. Supongamos ahora que todas las $\lfloor n / 5 \rfloor$ columnas, así como la fila del medio se ordenan, por arte de magia, con los elementos más pequeños situados en la parte superior y en el lado izquierdo respectivamente. La fila del medio corresponde a la matriz Z del algoritmo, y el elemento rodeado por un círculo correspondiendo a la mediana de esta matriz, que es el valor de p devuelto por el algoritmo. Claramente, cada uno de los elementos del cuadro es menor o igual que p. La conclusión se sigue automáticamente, puesto que el cuadro contiene aproximadamente entre tres quintos y la mitad de los elementos de T.

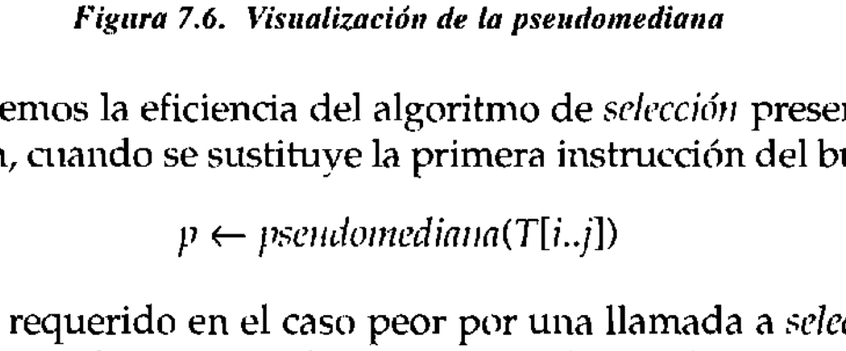


Figura 7.6. Visualización de la pseudomediana

Ahora analizaremos la eficiencia del algoritmo de *selección* presentado al principio de esta sección, cuando se sustituye la primera instrucción del bucle **repetir** por

$$p \leftarrow \textit{pseudomediana}(T[i..j])$$

Sea $t(n)$ el tiempo requerido en el caso peor por una llamada a *selección*($T[1..n]$, s). Consideremos i y j cualesquiera, tales que $1 \leq i \leq j \leq n$. El tiempo requerido para completar el bucle **repetir** con estos valores de i y j es esencialmente $t(m)$, donde $m = j - i + 1$ es el número de elementos que quedan por considerar. Cuando $n > 5$, el cálculo de *pseudomediana*(T) requiere un tiempo que está en $t(\lfloor n / 5 \rfloor) + O(n)$,

porque la matriz Z se puede construir en tiempo lineal, dado que cada llamada a *medianadhoc* requiere un tiempo constante. La llamada a *pivotebis* también requiere un tiempo lineal. Llegado este momento, o bien hemos terminado o bien hay que volver a pasar por el bucle con un máximo de $(7n + 12)/10$ elementos restantes por considerar. Por tanto, existe una constante d tal que

$$t(n) \leq dn + t(\lfloor n/5 \rfloor) + \max\{t(m) | m \leq (7n+12)/10\} \tag{7.6}$$

siempre y cuando $n > 5$.

La ecuación 7.1 no nos ayuda a resolver esta recurrencia, así que una vez más recurriremos a la inducción constructiva. Esta vez es precisa una cierta perspicacia incluso para hacer una suposición acerca de la respuesta en notación asintótica. (Recordemos que la propuesta evidente cuando analizábamos *quicksort* era $O(n \log n)$ porque ya habíamos analizado *ordenar por fusión*, y funcionaba; esta vez no tenemos tanta suerte.) Con una cierta experiencia, el hecho de que $\frac{1}{5} + \frac{7}{10} < 1$ es una indicación de que $t(n)$ puede muy bien ser lineal en n, lo cual es claramente lo mejor que podemos esperar; véase el problema 7.19.

Teorema 7.5.1 *El algoritmo de selección empleado con pseudomediana halla el s-ésimo elemento más pequeño entre n, con un tiempo que está en $\Theta(n)$ en el caso peor. En particular, la mediana se puede hallar en un tiempo lineal en el caso peor.*

Demostración. Sean $t(n)$ y d igual que antes. Claramente, $t(n) \in \Omega(n)$, porque el algoritmo tiene que examinar todos y cada uno de los elementos de T al menos una vez. Por tanto, queda por demostrar que $t(n) \in O(n)$. Postulemos la existencia de una constante c, desconocida por el momento, tal que $t(n) \leq cn$ para todo $n \geq 1$. Encontraremos un valor adecuado para esta constante en el proceso de demostración de su existencia por inducción matemática generalizada. También se utilizará la inducción constructiva para determinar la constante n_0 que separa el caso base del paso de inducción. Por el momento, nuestra única restricción es que $n_0 \geq 5$, porque la ecuación 7.6 solamente es aplicable cuando $n > 5$ (descubriremos que la elección evidente, $n_0 = 5$, no es válida).

◊ *Base*: consideremos cualquier entero n tal que $1 \leq n \leq n_0$. Tenemos que mostrar que $t(n) \leq cn$. Esto es sencillo, porque tenemos total libertad para seleccionar la constante c, y el número de casos es finito. Basta con seleccionar c al menos tan grande como $t(n)/n$. Por tanto, nuestra primera restricción sobre c es:

$$c \geq t(n)/n \text{ para todo } n \text{ tal que } 1 \leq n \leq n_0 \tag{7.7}$$

◊ *Paso de inducción*: consideremos cualquier entero $n > n_0$. Como hipótesis de inducción, supongamos que $t(m) \le cm$ cuando $1 \le m < n$. Deseamos restringir c de tal forma que $t(n) \le cn$ se siga de la hipótesis de inducción. Comenzando por la ecuación 7.6, y dado que $1 \le (7n + 12)/10$ cuando $n > n_0 \ge 5$:

$$\begin{aligned} t(n) &\le dn + t(\lfloor n/5 \rfloor) + \max\{t(m) \mid m \le (7n+12)/10\} \\ &\le dn + cn/5 + (7n+12)c/10 \text{ por la hipótesis de inducción} \\ &= 9cn/10 + dn + 6c/5 \\ &= cn - (c/10 - d - 6c/5n)n \end{aligned}$$

Se sigue que $t(n) \le cn$ siempre y cuando $c/10 - d - 6c/5n \ge 0$, lo cual es equivalente a $(1\text{-}12/n)c \ge 10d$. Esto es posible siempre y cuando sea $n \ge 13$ (para que $1 - 12/n > 0$), en cuyo caso c no debe de ser menor que $10d/(1 - 12/n)$.Teniendo en cuenta que $n > n_0$, cualquier elección de $n_0 \ge 12$ será aceptable, siempre y cuando c se seleccione en consecuencia. Más exactamente, todo va bien siempre y cuando $n_0 \ge 12$ y

$$c \ge \frac{10d}{1 - \frac{12}{n_0 + 1}} \tag{7.8}$$

que es nuestra segunda y última restricción sobre c y n_0. Por ejemplo, el paso de inducción es correcto si tomamos $n_0 = 12$ y $c \ge 130d$, o bien $n_0 = 23$ y $c \ge 20d$, o bien $n_0 = 131$ y $c \ge 11d$.

Reuniendo las restricciones dadas por las ecuaciones 7.7 y 7.8, y seleccionando $n_0 = 23$ para concretar, basta con hacer

$$c = \text{máx}(20d, \text{máx}\{t(m)/m \mid 1 \le m \le 23\})$$

para finalizar la demostración por inducción constructiva de que $t(n) \le cn$ para todo $n \ge 1$.

7.6 MULTIPLICACIÓN DE MATRICES

Sean A y B dos matrices $n \times n$ que hay que multiplicar, y sea C su producto. El algoritmo clásico de multiplicación de matrices proviene directamente de su definición:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Cada entrada de C se calcula en un tiempo que está en $\Theta(n)$, suponiendo que la suma y la multiplicación de escalares sean operaciones elementales. Dado que hay que calcular n^2 entradas, el producto AB se puede calcular en un tiempo que está en $\Theta(n^3)$.

Hacia el final de los años 60, Strassen dio lugar a un revuelo considerable, al mejorar este algoritmo. Desde un punto de vista algorítmico, este descubrimiento

fue un hito en la historia de «divide y vencerás», aun cuando el algoritmo no menos sorprendente para multiplicar enteros muy grandes (Sección 7.1) se hubiera descubierto casi una década antes. La idea básica del algoritmo de Strassen es similar a la anterior. En primer lugar, mostramos que las matrices 2×2 se pueden multiplicar empleando menos multiplicaciones escalares que las ocho que requiere aparentemente la definición. Sean

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \text{ y } B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

las dos matrices que hay que multiplicar. Consideremos las operaciones siguientes, cada una de las cuales necesita una única multiplicación:

$$\left.\begin{aligned} m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\ m_2 &= a_{11}b_{11} \\ m_3 &= a_{12}b_{21} \\ m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\ m_5 &= (a_{21} + a_{22})(b_{12} - b_{11}) \\ m_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\ m_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21}) \end{aligned}\right\} \tag{7.9}$$

Dejaremos al lector verificar que el producto matricial AB que se pide está dado por la matriz siguiente.

$$C = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix} \tag{7.10}$$

Por tanto, es posible multiplicar dos matrices 2×2 empleando solamente siete multiplicaciones escalares. A primera vista, este algoritmo no parece muy interesante: utiliza un elevado número de sumas y restas en comparación con las cuatro sumas que bastan en el algoritmo clásico.

Si ahora sustituimos cada entrada de A y B por una matriz $n \times n$, obtenemos un algoritmo que puede multiplicar dos matrices $2n \times 2n$ efectuando siete multiplicaciones de matrices $n \times n$, así como un cierto número de sumas y restas de matrices $n \times n$. Esto es posible porque el algoritmo 2×2 no se basa en la conmutatividad de la multiplicación escalar. Dado que es mucho más rápido sumar matrices grandes que multiplicarlas, el ahorro de una multiplicación compensa sobradamente las sumas adicionales.

Sea $t(n)$ el tiempo necesario para multiplicar dos matrices $n \times n$ mediante el uso recursivo de las ecuaciones 7.9 y 7.10. Supongamos por sencillez que n es una potencia de 2. Dado que las matrices se pueden sumar y restar en un tiempo que está en $\Theta(n^2)$, $t(n) = 7t(n/2) + g(n)$, en donde $g(n) \in \Theta(n^2)$. Esta recurrencia es otro ejemplo de nuestro análisis general de los algoritmos divide y vencerás. La ecuación 7.1 es aplicable con $c = 7$, $b = 2$ y $k = 2$. Dado que $c > b^k$, el tercer caso produce $t(n) \in \Theta(n^{\lg 7})$. Las matrices cuadradas cuyo tamaño no sea una potencia de 2 se pueden tratar fácilmente añadiéndoles filas y columnas de ceros, y doblando co-

mo mucho su tamaño, lo cual no afecta al tiempo asintótico de ejecución. Como $\lg 7 < 2{,}81$, resulta entonces posible multiplicar dos matrices en un tiempo que está en $O(n^{2.81})$, siempre y cuando las operaciones escalares sean elementales.

Siguiendo el descubrimiento de Strassen, un cierto número de investigadores ha intentado mejorar la constante ω de tal manera que sea posible multiplicar dos matrices $n \times n$ en un tiempo que esté en $O(n^{\omega})$. Lo primero que se podía intentar, evidentemente, era multiplicar dos matrices 2×2 empleando *seis* multiplicaciones escalares. Pero en 1971, Hopcroft y Kerr demostraron que esto era imposible cuando no se puede emplear la conmutatividad de la multiplicación. Lo siguiente que había que intentar era encontrar una forma de multiplicar dos matrices 3×3 empleando como máximo 21 multiplicaciones escalares. Esto daría lugar a un algoritmo recursivo para multiplicar matrices $n \times n$ en un tiempo que estaría en $\Theta(n^{\log_3 21})$, que es asintóticamente más rápido que el algoritmo de Strassen, porque $\log_3 21 < \log_2 7$. Desafortunadamente, esto también es imposible.

Pasó casi una década antes de que Pan descubriera una forma de multiplicar dos matrices 70×70 empleando 143.640 multiplicaciones escalares —compárese esto con las 343.000 que requiere el algoritmo clásico— y de hecho $\log_{70} 143.640$ es un poquitín más pequeño que $\lg 7$. Este descubrimiento dio pie a la denominada *guerra de los decimales*. Se fueron descubriendo sucesivamente numerosos algoritmos cada vez más eficientes asintóticamente. Por ejemplo, al final de 1979 se sabía que era posible multiplicar matrices en un tiempo que estaba en $O(n^{2,521813})$; imagine la excitación reinante en enero de 1980 cuando esto se mejoró hasta alcanzar $O(n^{2,521801})$. El algoritmo de multiplicación matricial asintóticamente más rápido conocido en el momento de escribir esto se remonta a 1986, año en que Coppersmith y Winograd descubrieron que es posible, al menos en teoría, multiplicar dos matrices $n \times n$ en un tiempo que está en $O(n^{2.376})$. Sin embargo, y como consecuencia de las constantes ocultas, ninguno de los algoritmos hallados después del de Strassen tiene una aplicación práctica real.

7.7 EXPONENCIACIÓN

Sean a y n dos enteros. Deseamos calcular la exponenciación $x = a^n$. Por sencillez, supondremos en toda esta sección que $n > 0$. Si n es pequeño, el algoritmo evidente resulta adecuado:

```
función exposec(a, n)
    r ← a
    para i ← 1 hasta n − 1 hacer r ← a × r
    devolver r
```

Este algoritmo requiere un tiempo que está en $\Theta(n)$, puesto que la instrucción $r \leftarrow a \times r$ se ejecuta exactamente $n - 1$ veces, *siempre y cuando las multiplicaciones cuenten como operaciones elementales*. Sin embargo, en la mayoría de las computadoras, hasta los valores pequeños de n y de a hacen que este algoritmo produzca desbordamientos enteros. Por ejemplo, 15^{17} no cabe en un entero de 64 *bits*.

Si deseamos manejar operandos más grandes, es preciso tener en cuenta el tiempo necesario para cada multiplicación. Sea $M(q, s)$ el tiempo necesario para multiplicar dos enteros de tamaños q y s. Para nuestros propósitos, no importa si consideramos el tamaño de los enteros en dígitos decimales, en *bits* o en cualquier otra base fija mayor que 1. Supongamos por sencillez que $q_1 \leq q_2$ y $s_1 \leq s_2$ implican que $M(q_1, s_1) \leq M(q_2, s_2)$. Estimemos cuánto tiempo invierte nuestro algoritmo multiplicando enteros cuando se invoca *exposec*(a, n). Sea m el tamaño de a. En primer lugar, observemos que el producto de dos enteros de tamaños i y j tiene un tamaño que es al menos $i + j - 1$ y como máximo $i + j$; véase el problema 7.24. Sean r_i y m_i el valor y el tamaño de r al principio de la i-ésima pasada por el bucle. Claramente, $r_1 = a$ y por tanto $m_1 = m$. Dado que $r_{i+1} = ar_i$, el tamaño de r_{i+1} es al menos $m + m_i - 1$ y como máximo $m + m_i$. La demostración por inducción matemática de que $im - i + 1 \leq m_i \leq im$ para todo i se sigue inmediatamente. Por tanto, la multiplicación efectuada en la i-ésima pasada por el bucle afecta a un entero de tamaño m y a un entero cuyo tamaño se encuentra entre $im - i + 1$ e im, lo cual requiere un tiempo que está entre $M(m, im - i + 1)$ y $M(m, im)$. El tiempo total $T(m, n)$ que se invierte en multiplicaciones cuando se calcula a^n con *exposec* es, por tanto:

$$\sum_{i=1}^{n-1} M(m, im - i + 1) \leq T(m,n) \leq \sum_{i=1}^{n} M(i, im) \tag{7.11}$$

donde m es el tamaño de a. $T(m,n)$ es una buena estimación del tiempo total requerido por *exposec*, puesto que la mayor parte del trabajo se invierte en realizar estas multiplicaciones. Si utilizamos el algoritmo clásico de multiplicación (Sección 1.2), entonces $M(q, s) \in \Theta(qs)$. Sea c tal que $M(q, s) \leq cqs$:

$$T(m,n) \leq \sum_{i=1}^{n-1} M(m, im) \leq \sum_{i=1}^{n-1} c \; m \; im$$

$$= cm^2 \sum_{i=1}^{n-1} i < cm^2 n^2$$

De esta forma, $T(m, n) \in O(m^2n^2)$. Resulta igualmente fácil demostrar, a partir de la ecuación 7.11, que $T(m, n) \in \Omega(m^2n^2)$ y por tanto que $T(m, n) \in \Theta(m^2n^2)$; véase el Problema 7.25. Por otra parte, si utilizamos el algoritmo de multiplicación de divide y vencerás que se describía anteriormente en este capítulo, $M(q,s) \in \Theta(sq^{\lg(3/2)})$ cuando $s \geq q$, y un argumento similar da $T(m,n) \in \Theta(n^{\lg 3}n^2)$.

La observación clave para mejorar *exposec* consiste en que $a^n = (a^{n/2})^2$ cuando n es par. Esto es interesante porque $a^{n/2}$ se puede calcular aproximadamente cuatro veces más deprisa que a^n con *exposec*, y basta una única elevación al cuadrado (que es una multiplicación) para obtener el resultado deseado a partir de $a^{n/2}$. Esto produce la recurrencia siguiente:

$$a^n = \begin{cases} a & \text{si } n = 1 \\ \left(a^{n/2}\right)^2 & \text{si } n \text{ es par} \\ a \times a^{n-1} & \text{en caso contrario} \end{cases}$$

Por ejemplo,

$$a^{29} = a\, a^{28} = a(a^{14})^2 = a((a^7)^2)^2 = \cdots = a\ ((a(a\, a^2)^2)^2)^2$$

que sólo utiliza tres multiplicaciones y cuatro elevaciones al cuadrado en lugar de las 28 multiplicaciones que se necesitaban con *exposec*. La recurrencia anterior da lugar al siguiente algoritmo:

función *expoDV*(a, n)
  si $n = 1$ **entonces devolver** a
  si n es par **entonces devolver** $[expoDV(a, n/2)]^2$
  devolver $a \times expoDV(a, n-1)$

Para analizar la eficiencia de este algoritmo, nos concentraremos primero en el *número* de multiplicaciones (contando la elevación al cuadrado como una multiplicación) efectuadas por una llamada a *expoDV*(a, n). Observemos que el flujo de control del algoritmo no depende del valor de a, y por tanto el número de multiplicaciones sólo es una función del exponente n; lo denotaremos $N(n)$.

Cuando $n = 1$ no se efectúa ninguna multiplicación, así que $N(1) = 0$. Cuando n es par, se efectúa una multiplicación (la elevación al cuadrado de $a^{n/2}$) además de las $N(n/2)$ multiplicaciones implicadas en la llamada recursiva a *expoDV*(a, $n/2$). Cuando n es impar, se efectúa una multiplicación (la de a por a^{n-1}) además de las $N(n-1)$ multiplicaciones requeridas por la llamada recursiva a *expoDV*(a, $n-1$). Entonces tenemos la recurrencia siguiente.

$$N(n) = \begin{cases} 0 & \text{si } n = 1 \\ N(n/2) + 1 & \text{si } n \text{ es par} \\ N(n-1) + 1 & \text{en caso contrario} \end{cases} \qquad (7.12)$$

Una característica poco habitual de esta recurrencia es que no da lugar a una función creciente de n. Por ejemplo:

$$N(15) = N(14) + 1 = N(7) + 2 = N(6) + 3 = N(3) + 4 = N(2) + 5$$

$$= N(1) + 6 = 6$$

mientras que $N(16) = N(8) + 1 = N(4) + 2 = N(2) + 3 = N(1) + 4 = 4$.

Esta función no es ni siquiera *eventualmente* no decreciente; véase el Problema 7.26. Por tanto, no se puede utilizar directamente la ecuación 7.1.

Para tratar esta recurrencia, es útil acotar superior e inferiormente la función mediante funciones no decrecientes. Cuando $n > 1$ es impar:

$$N(n) = N(n-1) + 1 = N((n-1)/2) + 2 = N(\lfloor n/2 \rfloor) + 2$$

Por otra parte, cuando n es par, $N(n) = N(\lfloor n/2 \rfloor)+1$, puesto que $\lfloor n/2 \rfloor = n/2$ en ese caso. Por tanto,

$$N(\lfloor n/2 \rfloor) + 1 \leq N(n) \leq N(\lfloor n/2 \rfloor) + 2 \tag{7.13}$$

para todo $n > 1$. Sean N_1 y N_2 dos funciones definidas mediante

$$N_i(n) = \begin{cases} 0 & \text{si } n = 1 \\ N_i(\lfloor n/2 \rfloor) + i & \text{en caso contrario} \end{cases} \tag{7.14}$$

para $i = 1$ e $i = 2$. Empleando la ecuación 7.13, es fácil comprobar por inducción matemática que $N_1(n) \leq N_2(n)$ para todo n; véase el Problema 7.27. Pero ahora tanto N_1 como N_2 son no decrecientes (también es fácil demostrarlo por inducción matemática). Además, $\lfloor n/2 \rfloor = n/2$ cuando $n > 1$ es una potencia de 2. La ecuación 7.1 es aplicable a $N_1(n)$ y a $N_2(n)$ con $c = 1$, $b = 2$ y $k = 0$, para dar que estas dos funciones están en $\Theta(\log n)$. Concluimos que $N(n)$ se encuentra también en $\Theta(\log n)$. Recordemos que *exposec* requería un número de multiplicaciones que estaba en $\Theta(n)$ para efectuar la misma exponenciación. Por tanto, *expoDV* es más eficiente que su competidor empleando este criterio. Queda por ver si también es significativamente mejor cuando se tiene en cuenta el *tiempo* invertido en esas multiplicaciones.

Sea una vez más $M(q, s)$ el tiempo necesario para multiplicar dos enteros de tamaños q y s, y denotemos ahora mediante $T(m, n)$ el tiempo que invierte en multiplicar una llamada a *expoDV*(a, n), en donde m es el tamaño de a. Recordemos que el tamaño de a^i es como mucho im. La inspección del algoritmo *expoDV* da lugar a la recurrencia siguiente:

$$T(m,n) \leq \begin{cases} 0 & \text{si } n = 1 \\ T(m, n/2) + M(mn/2, mn/2) & \text{si } n \text{ es par} \\ T(m, n-1) + M(m, (n-1)m) & \text{en caso contrario} \end{cases} \tag{7.15}$$

Como en la recurrencia para N, esto implica que

$$T(m,n) \leq T(m, \lfloor n/2 \rfloor) + M(m\lfloor n/2 \rfloor, m\lfloor n/2 \rfloor) + M(m, (n-1)m)$$

para todo $n > 1$. Si $M(q,s) \in \Theta(sq^{\alpha-1})$ para alguna constante α cuando $s \geq q$ ($\alpha = 2$ con el algoritmo clásico de multiplicación, mientras que $\alpha = \lg 3$ con multiplicación por divide y vencerás), el problema 7.29 muestra que $T(m,n) \in O(m^\alpha n^\alpha)$. La cota inferior equivalente es más fácil de obtener. La última multiplicación efectuada por *expoDV*(a, *n*), o la penúltima, dependiendo de si *n* es par o impar, consiste en elevar al cuadrado $a^{\lfloor n/2 \rfloor}$, que es un entero de tamaño mayor que $(m-1)\lfloor n/2 \rfloor$. Esto cuesta como mínimo $M((m-1)\lfloor n/2 \rfloor, (m-1)\lfloor n/2 \rfloor)$, que se encuentra en $\Omega(((m-1)\lfloor n/2 \rfloor)^\alpha)$, y por tanto en $\Omega(m^\alpha n^\alpha)$. Concluimos que $T(m,n) \in \Theta(m^\alpha n^\alpha)$.

Para resumir, la tabla siguiente nos da el tiempo necesario para calcular a^n, en donde *m* es el tamaño de *a*, dependiendo de si utilizamos *exposec* o *expoDV*, y de si utilizamos el algoritmo de multiplicación clásico o el de divide y vencerás (D y V).

	multiplicación clásica	multiplicación *D* y *V*
exposec	$\Theta(m^2n^2)$	$\Theta(m^{\lg 3}n^2)$
expoDV	$\Theta(m^2n^2)$	$\Theta(m^{\lg 3}n^{\lg 3})$

Resulta interesante constatar que no ganamos nada —salvo quizá un factor constante de velocidad— al ser sólo medio despiertos: tanto *exposec* como *expoDV* requieren un tiempo que está en $\Theta(m^2n^2)$ cuando se utiliza el algoritmo clásico de multiplicación. Esto es cierto aun cuando *expoDV* utiliza una cantidad de multiplicaciones exponencialmente menor que *exposec*.

Tal como sucede en la búsqueda binaria, el algoritmo *expoDV* requiere solamente una llamada recursiva en un caso de menor tamaño. Por tanto, es un ejemplo de simplificación, más que de divide y vencerás. Sin embargo, esta llamada recursiva no se encuentra dinámicamente al final del algoritmo, puesto que después de ella queda por hacer una multiplicación. Esto hace que sea más difícil encontrar una versión iterativa. Sin embargo, tal algoritmo existe y se corresponde intuitivamente con calcular $a^{29} = a^1a^4a^8a^{16}$. Lo indicamos sin más dilación, observando que esta versión sencilla hace sistemáticamente dos multiplicaciones inútiles, incluyendo la última elevación al cuadrado de *x*.

```
función expoiter(a, n)
    i ← n; r ← 1; x ← a
    mientras i > 0 hacer
        si i es impar entonces r ← rx
        x ← x²
        i ← i ÷ 2
    devolver r
```

7.8 ENSAMBLANDO TODAS LAS PIEZAS: INTRODUCCIÓN A LA CRIPTOGRAFÍA

Retrotrayéndonos a la sección anterior, resulta descorazonador que una reducción exponencial del número de multiplicaciones necesario para calcular a^n no se traduzca en un espectacular ahorro de tiempo de ejecución. Sin embargo, existen aplicaciones para las cuales *es* razonable contar todas las multiplicaciones como de un mismo coste. Tal es el caso si estamos interesados en la aritmética *modular*, esto es, en el cálculo de a^n módulo algún tercer entero z. Recuerde que x mod z denota el resto de la división entera de x por z. Por ejemplo, 25 mod 7 = 4 porque 25 = 3×7 + 4. Si x e y son dos enteros entre 0 y $z-1$, y si z es un entero de tamaño m, entonces la multiplicación modular xy mod z necesita una multiplicación entera ordinaria de dos enteros de tamaño m como máximo, dando lugar a un entero de tamaño $2m$ como máximo, seguida por división del producto por z, un entero de tamaño m, para calcular el resto de la división. Por tanto, el tiempo que requiere cada multiplicación modular es más bien independiente de los dos números involucrados. Utilizaremos dos propiedades elementales de la aritmética modular; véase el Problema 7.30.

$$xy \bmod z = [(x \bmod z) \times (y \bmod z)] \bmod z$$

y

$$(x \bmod z)^y \bmod z = x^y \bmod z$$

Por tanto, *exposec*, *expoDV* y *expoiter* se pueden adaptar para calcular a^n mod z en aritmética modular, sin tener que manipular nunca enteros que sean más grandes que máx(a,z^2). Para ello, basta reducir módulo z después de cada multiplicación. Por ejemplo, *expoiter* da lugar al algoritmo siguiente:

```
función expomod(a, n, z)
    {Calcula a^n mod z}
    mientras i > 0 hacer
        si i es impar entonces r ← a mod z
        x ← x^2 mod z
        i ← i ÷ 2
    devolver r
```

El análisis de la sección anterior se aplica *mutatis mutandis* para concluir que este algoritmo requiere solamente un número de multiplicaciones modulares que está en $\Theta(\log n)$ para calcular a^n mod z. Un análisis más preciso muestra que el número de multiplicaciones modulares es igual al número de *bits* que hay en la expansión binaria de n, más el número de estos *bits* que son iguales a 1; por tanto, es aproximadamente igual a $\frac{3}{2}\lg n$ para un n típico. Por contraste, el algoritmo correspondiente a *exposec* requiere $n-1$ multiplicaciones de éstas para todo n. Para

concretar, supongamos que se desea calcular a^n mod z, en donde a, n y z son números de 200 dígitos, y supongamos también que los números de este tamaño se pueden multiplicar módulo z en un milisegundo. Nuestro algoritmo *expomod* calculará a^n mod z en menos de un segundo. El algoritmo correspondiente a *exposec* requeriría aproximadamente ¡179 veces la edad del Universo para la misma tarea!

Con todo lo impresionante que resulta esto, quizá se pregunte quién podría tener la necesidad de calcular exponenciaciones modulares tan enormes en la vida real. Resulta que la *criptografía* moderna, que es el arte y la ciencia de la comunicación secreta a través de canales poco seguros, depende crucialmente de ella. Consideremos dos partes que llamaremos Alicia y Benito, y supongamos que Alicia desea enviar un mensaje secreto m a Benito, a través de un canal que podría estar sometido a escuchas. Para evitar que otras personas lean el mensaje, Alicia lo transforma en un *texto cifrado* c, que envía a Benito. Esta transformación es el resultado de un algoritmo de *cifrado* cuya salida depende no solamente del mensaje m sino también de otro parámetro k conocido con el nombre de *clave*. Clásicamente, esta clave es información secreta, que debe haber sido acordada entre Alicia y Benito antes de que pueda tener lugar una conversación secreta. A partir de c y de su conocimiento de k, Benito puede reconstruir el verdadero mensaje m de Alicia. Estos sistemas de criptografía confían en que el escucha que intercepte c pero no conozca k no sea capaz de determinar m a partir de la información disponible.

Esta aproximación a la criptografía ha sido utilizada con más o menos éxito a lo largo de la Historia. Su requisito es que las partes deban compartir información secreta antes que la comunicación pueda resultar aceptable para militares y diplomáticos, pero no para el ciudadano ordinario. En la era de las superautopistas electrónicas, es deseable para dos ciudadanos cualesquiera poder comunicarse en privado sin coordinación previa. ¿Pueden comunicarse Alicia y Benito secretamente ante la vista de una tercera parte, si no comparten un secreto antes de establecer la comunicación? La era de la *criptografía de clave pública* nació cuando la idea de que esto pudiera ser posible fue desarrollada por Diffie, Hellman y Merkle a mediados de los años setenta. Aquí presentamos la sorprendentemente sencilla solución que descubrieron pocos años después Rivest, Shamir y Adleman, y que recibió el nombre de *sistema criptográfico RSA* a partir de los nombres de sus inventores.

Consideremos dos números primos p y q de 100 dígitos, seleccionados aleatoriamente por Benito; véase la Sección 10.6.2 para un algoritmo eficiente capaz de comprobar la primalidad de números tan grandes. Sea z el producto de p y q. Benito puede calcular eficientemente z a partir de p y q. Sin embargo, ningún algoritmo conocido puede recalcular p y q a partir de z dentro del tiempo de duración del Universo, ni siquiera empleando la computadora más rápida conocida en el momento de escribir esto. Sea ϕ igual a $(p-1)(q-1)$. Sea n un entero seleccionado aleatoriamente por Benito entre 1 y $z-1$, que no tenga factores comunes con ϕ. (No es necesario que Benito compruebe explícitamente el que n tenga la propiedad deseada, porque lo averiguará en seguida de no ser así.) La teoría elemental de números nos dice que existe un único entero s entre 1 y $z-1$ tal que ns mod

$\varnothing = 1$. Además, s es fácil de calcular a partir de n y $\varnothing$ —véase el Problema 7.31— y su existencia es la demostración de que n y $\varnothing$ no poseen factores comunes. Si s no existe, Benito tiene que seleccionar aleatoriamente un nuevo valor para n; todos los intentos tienen buenas probabilidades de éxito. El teorema clave es que $a^n \bmod z = a$ siempre que $0 \le a < z$ y $x \bmod \varnothing = 1$.

Para permitir que Alicia o alguien más se comunique privadamente con él, Benito hace pública su selección de z y n, pero mantiene secreto el valor de s. Sea m un mensaje que Alicia desea transmitir a Benito. Empleando una codificación estándar como ASCII, Alicia transforma su mensaje en una sucesión de *bits*, que interpreta como un número a. Supongamos por sencillez que $0 \le a \le z - 1$; en caso contrario, Alicia puede cortar su mensaje m en trozos de tamaño adecuado. A continuación, Alicia utiliza el algoritmo *expomod* para calcular $c = a^n \bmod z$, que envía a Benito a través de un canal sin protección. Empleando su conocimiento privado de s, Benito calcula a, y por tanto el mensaje m de Alicia, mediante una llamada a *expomod*(c, s, z). Esto funciona porque

$$c^s \bmod z = \left(a^n \bmod z\right)^s \bmod z = \left(a^n\right)^s \bmod z = a^{ns} \bmod z = a.$$

Consideremos ahora la tarea del escucha. Suponiendo que haya interceptado todas las comunicaciones entre Alicia y Benito, el escucha conoce z, n y c. Su objetivo es determinar el mensaje a de Alicia, que es el único número entre 0 y $z - 1$ tal que $c = a^n \bmod z$. Entonces tiene que calcular la *raíz* n-ésima de c módulo z. Para este cálculo no se conoce un algoritmo eficiente: las exponenciaciones modulares se pueden calcular eficientemente con *expomod*, pero parece que el proceso inverso no es factible. El mejor método que se conoce en la actualidad es el evidente: factorizar z en p y q, calcular $\varnothing$ como $(p-1)(q-1)$, utilizar el problema 7.31 para calcular s a partir de n y $\varnothing$, y calcular $a = c^n \bmod z$ exactamente igual que lo habría hecho Benito. Todos los pasos de este ataque son factibles, menos el primero: la factorización de un número de 200 dígitos está fuera del alcance de la tecnología actual. Por tanto, la ventaja de Benito para descifrar mensajes destinados a él surge del hecho de que solamente él conoce los factores de z, que son necesarios para calcular $\varnothing$ y s. Este conocimiento no proviene de sus capacidades de factorización, sino más bien del hecho de que ha seleccionado primero los factores de z, y ha calculado z a partir de ellos.

En el momento de escribir esto, la seguridad de este esquema criptográfico no se ha establecido matemáticamente: la factorización puede resultar ser fácil, o puede incluso ser innecesaria, para violar este sistema. Además, se *conoce* un algoritmo de factorización eficiente, pero requiere la existencia de una *computadora cuántica*, dispositivo cuya construcción va más allá de los límites de la tecnología actual; véase la Sección 12.6. Sin embargo, el sistema secreto que acabamos de describir goza de amplia aceptación como una de las mejores invenciones en la historia de la criptografía.

7.9 PROBLEMAS

Problema 7.1. Consideremos un algoritmo cuyo tiempo de ejecución $t(n)$ en ejemplares de tamaño n es tal que $t(n) = 3t(n/2) + g(n)$ cuando n es par y suficientemente grande, donde $g(n) \in O(n)$. Esta es la recurrencia que hallábamos anteriormente en nuestro estudio del algoritmo divide y vencerás para multiplicar enteros muy grandes en la Sección 7.1, antes de haber discutido la forma de manejar operandos de longitud impar. Recordemos que al resolverla se tiene $t(n) \in \Theta(n^{\lg 3} \mid n$ es una potencia de 2). Dado que $t(n) = 3t(n/2) + g(n)$ es válida para todos los valores *pares* de n suficientemente grandes, y no sólo cuando n es meramente una potencia de 2, puede resultar tentador concluir que $t(n) \in \Theta(n^{\lg 3} \mid n$ es par. Demostrar que esta conclusión podría ser prematura sin más información acerca del comportamiento de $t(n)$ cuando n es impar. Por otra parte, dar una condición sencilla y natural sobre $t(n)$ que permitiría la conclusión de que está en $\Theta(n^{\lg 3})$ incondicionalmente.

Problema 7.2. En la Sección 7.1 vimos un algoritmo divide y vencerás para multiplicar dos enteros de n cifras en un tiempo que estaba en $\Theta(n^{\lg 3})$. La idea clave era reducir las multiplicaciones necesarias a tres multiplicaciones de números de tamaño mitad. Demostrar que los operandos que hay que multiplicar se pueden separar en tres partes en lugar de hacerlo en dos, para obtener el producto requerido tras cinco multiplicaciones de enteros de tamaño aproximado $n/3$, en lugar de las nueve submultiplicaciones que podrían parecer necesarias a primera vista. Analizar la eficiencia del algoritmo de divide y vencerás sugerido por esta idea. ¿Es mejor que nuestro algoritmo de la Sección 7.1?

Problema 7.3. Generalizar el algoritmo sugerido en el problema 7.2, mostrando que la multiplicación de dos enteros de n cifras se puede reducir a $2k-1$ multiplicaciones de enteros aproximadamente k veces más cortos, para toda constante entera k. Concluir que existe un algoritmo A_α que puede multiplicar dos enteros de n cifras en un tiempo que está en $O(n^\alpha)$ para todo número real $\alpha > 1$.

Problema 7.4. Utilizar un argumento sencillo para demostrar que el Problema 7.3 sería imposible si exigiese que el algoritmo A_α requiriera un tiempo en $\Omega(n^\alpha)$.

Problema 7.5. Continuando con el Problema 7.3, consideremos el siguiente algoritmo para multiplicar enteros muy grandes.

```
función supermul(u, v)
    {Por sencillez, suponemos que u y v
    son del mismo tamaño}
    n ← tamaño de u y v
    α ← 1 + (lg lg n)/lg n
    devolver A_α(u,v)
```

A primera vista este algoritmo parece multiplicar dos enteros de n cifras en un tiempo que está en $O(n \lg n)$ puesto que $n^\alpha = n \lg n$ cuando $\alpha = 1 + (\lg \lg n)/\lg n$. Buscar al menos dos errores fundamentales en este análisis de *supermul*.

Problema 7.6. Si todavía no ha resuelto los Problemas 4.6, 4.7 y 4.8, ¡éste es el momento!

Problema 7.7. ¿Qué sucede con la eficiencia de los algoritmos de divide y vencerás si en lugar de utilizar un umbral para decidir cuando hay que volver al subalgoritmo básico, empleamos la recurrencia un máximo de r veces, para alguna constante r, y utilizamos después el subalgoritmo básico?

Problema 7.8. Sean a y b constantes reales y positivas. Para cada número real y positivo

s, consideremos la función $f_s: \mathbb{R}^{+0} \to \mathbb{R}^{+0}$ definida por

$$f_s(x) = \begin{cases} ax^2 & \text{si } x \leq s \\ 3f_s(x/2) + bx & \text{en caso contrario} \end{cases}$$

Demostrar por inducción matemática que si $u = 4b/a$ y v es una constante real y positiva arbitraria, entonces $f_u(x) \leq f_v(x)$ para todo número real positivo x.

Nota: La constante u se ha seleccionado de tal manera que $au^2 = 3a(u/2)^2 + bu$. Este problema ilustra la regla según la cual el umbral óptimo en un algoritmo de divide y vencerás se puede estimar hallando el tamaño n del ejemplar para el cual no hay diferencia entre aplicar directamente el subalgoritmo básico o pasar a un nivel más de recursión. Las cosas no son tan fáciles en la práctica, porque el tamaño de los subcasos no se puede dividir por dos indefinidamente, pero con todo esta regla sigue siendo una guía excelente.

Problema 7.9. Consideremos el algoritmo de búsqueda *secuencial* de la Sección 7.3 que busca algún elemento x en una matriz $T[1..n]$. Suponiendo que los elementos de T sean diferentes, que x esté realmente en la matriz, y que se pueda encontrar en cualquiera de las posiciones con igual probabilidad, demostrar que el número medio de pasadas por el bucle es de $(n + 1)/2$.

Problema 7.10. El algoritmo recursivo *busquedabin* para hacer búsquedas binarias (Sección 7.3) trata la búsqueda en una matriz vacía $T[1..0]$ como un caso especial (se comprueba explícitamente si $n = 0$). Convénzase a sí mismo de que esta comprobación no es necesaria en la versión iterativa *biniter*, independientemente del resultado de la comparación entre x y $T[0]$, siempre y cuando esté permitido consultar el valor de $T[0]$ (lo cual quiere decir que no debe de haber comprobaciones de rango).

Problema 7.11. Una rápida inspección de algoritmo iterativo de búsqueda binaria *binia* de la Sección 7.3 muestra lo que parece ser un ineficiencia. Supongamos que T contiene 17 elementos diferentes, y que es $x = T[13]$. En la primera pasada por el bucle, $i = 1, j = 17$ y $k = 9$. La comparación entre x y $T[9]$ da lugar a la asignación $i \leftarrow 10$. En la segunda pasada por el bucle, $i = 10$, $j = 17$ y $k = 13$. Entonces se hace una comparación entre x y $T[13]$. Esta comparación podría permitirnos finalizar inmediatamente la búsqueda, pero no se hace una comprobación de igualdad, y por tanto se efectúa la asignación $j \leftarrow 13$. Hacen falta dos pasadas más por el bucle antes de salir con $i = j = 13$. Por comparación, el algoritmo *búsqueda binaria* de la Sección 4.2.4 abandona el bucle inmediatamente después de encontrar el elemento que estaba buscando.

Por tanto, *biniter* efectúa sistemáticamente un número de pasadas por el bucle que está en $\Theta(\log n)$, independientemente de la posición de x en T, mientras que *búsqueda binaria* puede hacer sólo una o dos pasadas por el bucle si x está en posición favorable. Por otra parte, una pasada por el bucle en *búsqueda binaria* tarda un poquito más en ejecutar el análisis en el caso medio que una pasada por el bucle en *biniter*. Para determinar qué algoritmo es mejor asintóticamente, analice con precisión el número medio de pasadas por el bucle que hace cada uno de ellos. Por sencillez, suponga que T contiene n elementos diferentes y que x aparece en T, ocupando cada posición con igual probabilidad. Demuestre la existencia de una constante c tal que el análisis *búsqueda binaria* ahorra en el caso promedio como mucho c pasadas por el bucle en comparación con *biniter*. En conclusión, ¿cuál de los algoritmos es mejor cuando el caso es arbitrariamente grande?

Problema 7.12. Sea $T[1..n]$ una matriz ordenada formada por enteros diferentes, algu-

nos de los cuales pueden ser negativos. De un algoritmo que pueda hallar un índice i tal que $1 \leq i \leq n$ y $T[i] = i$, siempre y cuando exista el tal índice. El algoritmo debe de necesitar un tiempo que esté en $O(\log n)$ en el caso peor.

Problema 7.13. El uso de centinelas en el algoritmo *fusionar* requiere la disponibilidad de una celda adicional en las matrices que haya que fusionar; véase la Sección 7.4.1. Aun cuando esto no importa cuando se utiliza *fusionar* dentro de *ordenarporfusión*, puede ser un inconveniente en otras aplicaciones. Lo que es más importante, nuestro algoritmo de fusión puede fallar si no es posible garantizar que los centinelas sean estrictamente mayores que todo posible valor de las matrices que hay que fusionar.

(a) Dar un ejemplo de matrices U y V que estén ordenadas, pero tales que el resultado de *fusionar*(U, V, T) no es el que debería ser. ¿Cuál es el contenido de T después de esta llamada patológica? (Se admite el valor ∞ en las matrices U y V, y quizá sea necesario especificar valores de U y V que estén fuera de los límites de las matrices.)
(b) Dar un **procedimiento** para fusionar matrices que no utilice centinelas. Su algoritmo debe de funcionar correctamente en un tiempo lineal, siempre y cuando las matrices U y V estén ordenadas antes de la llamada.

Problema 7.14. En la Sección 7.4.1 vimos un algoritmo *fusionar* capaz de fusionar dos matrices ordenadas U y V en un tiempo lineal, esto es, en un tiempo que está en el orden exacto de la suma de las longitudes de U y V. Obtener otro algoritmo de fusión que consiga el mismo objetivo, también en tiempo lineal, pero sin utilizar una matriz auxiliar: las secciones $T[1..k]$ y $T[k+1..n]$ de la matriz se ordenan independientemente, y hay que ordenar toda la matriz $T[1..n]$ empleando solamente una cantidad fija de espacio adicional.

Problema 7.15. En lugar de descomponer $T[1..n]$ en dos matrices de tamaño mitad a efectos de ordenarla por fusión, podríamos pensar en partirla en tres matrices de tamaño $n \div 3$, $(n + 1) \div 3$ y $(n + 2) \div 3$, para ordenar recursivamente cada una de éstas, y fusionar entonces las tres matrices ordenadas. Proporcione una descripción más formal de este algoritmo, y analice su tiempo de ejecución.

Problema 7.16. Consideremos una matriz $T[1..n]$. Tal como en el análisis del caso promedio para *quicksort*, suponga que los elementos de T son diferentes, y que cada una de las $n!$ permutaciones iniciales distintas de los elementos es igualmente probable. Consideremos una llamada a *pivote*($T[1..n]$, l). Demuestre que las $(l-1)!$ permutaciones posibles de los elementos de $T[1..l-1]$ son igualmente probables después de la llamada. Demuestre una afirmación similar acerca de $T[l+1..n]$

Problema 7.17. Desarrollar un algoritmo de tiempo lineal para implementar el algoritmo *pivotebis* en las Secciones 7.4.2 y 7.5. El algoritmo debe de recorrer la matriz una sola vez, y no se pueden utilizar matrices auxiliares.

Problema 7.18. Demostrar que el algoritmo de selección de la Sección 7.5 requiere un tiempo lineal de análisis en el caso promedio si sustituimos la primera instrucción del bucle **repetir** por «$p \leftarrow T[i]$». Suponga que los elementos de la matriz son diferentes y que todas las permutaciones iniciales posibles son igualmente probables.

Problema 7.19. Sean $a_1, a_2, \ldots, a_k$ números reales positivos cuya suma sea estrictamente menor que 1. Considere una función $f: \mathbb{N} \rightarrow \mathbb{R}^{>0}$ tal que

$$f(n) \le f(\lfloor a_1 n \rfloor) + f(\lfloor a_2 n \rfloor) + \ldots + f(\lfloor a_k n \rfloor) + cn$$

para algún c positivo y para todo n suficientemente grande. Demostrar por inducción constructiva que $f(n) \in O(n)$.

¿Sería esto válido en general si la suma de los a_i fuera exactamente 1? Justifique su respuesta con un argumento sencillo.

Problema 7.20. Una matriz T contiene n elementos. Se desea encontrar los m elementos más pequeños, en donde m es mucho menor que n. ¿Qué haría usted?

(a) Ordenar T y tomar los m primeros elementos,
(b) Llamar a *selección* (T,i) para $i = 1, 2, \ldots, m$, o
(c) utilizar algún otro método.

Justifique su respuesta.

Problema 7.21. La matriz T es igual que la del problema anterior, pero ahora se necesitan los elementos de rango $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil + 1, \ldots \lceil n/2 \rceil + m - 1$. ¿Qué haría usted?

(a) Ordenar T y tomar los elementos adecuados
(b) Llamar a *selección* m veces, o
(c) Utilizar algún otro método.

Justifique su respuesta.

Problema 7.22. El número de sumas y restas que se necesitan para calcular el producto de dos matrices 2×2 empleando las ecuaciones 7.9 y 7.10 parece ser 24 a primera vista. Demuestre que esto se puede reducir a 15 empleando variables auxiliares para evitar volver a calcular términos tales como $m_1 + m_2 + m_4$.

Problema 7.23. Suponiendo que n sea una potencia de 2, hallar el número exacto de sumas y multiplicaciones escalares que necesita el algoritmo de Strassen para multiplicar dos matrices $n \times n$ (utilice el resultado del problema 7.22). Su respuesta dependerá del umbral empleado para dejar de hacer llamadas recursivas. Teniendo en cuenta lo aprendido en la Sección 7.2, proponga un umbral que minimice el número de operaciones escalares.

Problema 7.24. Diremos que un entero x tiene un tamaño (decimal) n es $10^{n-1} \le x \le 10^n - 1$. Demuestre que el producto de dos enteros i y j es de tamaño $i + j - 1$ como mínimo, y como máximo de tamaño $i + j$. Demuestre que esta regla es igualmente aplicable para cualquier base $b \ge 2$, en donde diremos que un entero x es de tamaño n si $b^{n-1} \le x \le b^n - 1$.

Problema 7.25. Sea $T(m, n)$ el tiempo invertido en multiplicaciones al calcular a^n mediante una llamada a *exposec*$/(a, n)$, donde m es el tamaño de a; véase la Sección 7.7. Utilice la ecuación 7.11 con $M(q, s) \in \Theta(qs)$ para concluir que $T(m, n) \in \Omega(m^2 n^2)$.

Problema 7.26. Consideremos la función $N(n)$ dada por la ecuación 7.12, que cuenta el número de multiplicaciones necesarias para calcular a^n mediante el algoritmo *expoDV* de la Sección 7.7. Vimos que era $N(15) > N(16)$. Demostrar la existencia de una infinidad de enteros n tales que $N(n) > N(n+1)$. Concluya que esta función no es eventualmente no decreciente.

Problema 7.27. Utilice las ecuaciones 7.12, 7.13 y 7.14 para demostrar que

$$N_1(n) \le N(n) \le N_2(n)$$

para todo n, y que tanto $N_1(n)$ como $N_2(n)$ son funciones no decrecientes.

Problema 7.28. El algoritmo *expoDV* de la Sección 7.7 no minimiza siempre las multipli-

caciones —incluyendo elevaciones al cuadrado— necesarias para calcular a^n. Por ejemplo, calcula a^{15} en la forma $a(a(aa^2)^2)^2$, que requiere seis multiplicaciones. Demuestre que lo cierto es que a^{15} se puede calcular con sólo cinco multiplicaciones. Resista la tentación de utilizar la fórmula $a^{15} = ((((a^2)^2)^2)^2) / a$ ¡afirmando después que la división es otra clase de multiplicación!

Problema 7.29. Consideremos el $T(m, n)$ dado por la Ecuación 7.15. Se trata del tiempo que se invierte multiplicando cuando se hace una llamada a *expoDV*(a, n), donde m es el tamaño de a; véase la Sección 7.7. Si $M(q,s) \in \Theta(sq^{\alpha-1})$ para alguna constante α cuando $s \geq q$, demostrar que

$$T(m,n) \in O\left(m^{\alpha} n^{\alpha}\right)$$

Problema 7.30. Considere tres enteros cualesquiera x, y y z tales que z es positivo. Demostrar que

$$xy \bmod z = [(x \bmod z) \times (y \bmod z)] \bmod z$$

y que

$$(x \bmod z)^y \bmod z = x^y \bmod z.$$

Sugerencia: Escriba x en la forma $qz + r$, en donde $r = x \bmod z$ y $q = x \div z$.

Problema 7.31. Sean u y v dos enteros positivos y sea d su máximo común divisor.

(a) Demostrar que existen dos enteros a y b tales que $au + bv = d$.

Sugerencia: Suponga sin pérdida de generalidad que $u \geq v$. Si $u = v$, entonces $d = v$ y el resultado es inmediato (haga $a = 0$ y $b = 1$). En caso contrario, sea $w = u \bmod v$. Observe que $v < u$ y que $w < v$. Demuestre primero que d es también el máximo común divisor de v y w, razón por la cual el algoritmo de Euclides calcula correctamente el máximo común divisor. Por inducción matemática, sean ahora a' y b' tales que $a'v + b'w = d$. Por último, haga $a = b'$ y $b = a' - (u \div v)b'$. Queda por demostrar que $au + bv = d$, que es lo que se deseaba.

(b) Dar un algoritmo eficiente para calcular d, a y b a partir de u y v. El algoritmo no debería calcular d antes de empezar a trabajar con a y b.

Sugerencia: La pista anterior es relevante.

(c) Considere dos enteros n y $\varnothing$ tales que $\text{mcd}(n, \varnothing) = 1$. Dar un algoritmo eficiente para determinar un entero s tal que $ns \bmod \varnothing = 1$.

Sugerencia: Utilizando el subproblema anterior, calcular s y t tales que $ns + t\varnothing = 1$.

Problema 7.32. En este problema le invitamos a que resuelva un ejemplo sencillo de cifrado y descifrado empleando el sistema criptográfico de clave pública RSA; véase la Sección 7.8. Suponga que Benito decide que sus dos primos «muy grandes» sean $p = 19$ y $q = 23$. Los multiplica, para obtener un $z = 437$. A continuación, selecciona aleatoriamente un $n = 13$. Calcule $\varnothing = (p - 1)(q - 1)$ y utilice el Problema 7.31 para hallar el único s entre 1 y $z \times 1$ tal que $ns \bmod \varnothing = 1$. Benito publica z y n, pero mantiene s en secreto.

A continuación, supongamos que Alicia desea enviar el mensaje sin cifrar $m = 123$ a Benito. Entonces busca la $z = 437$ y la $n = 13$ de Benito en el directorio público. Utilice *expomod* para calcular el mensaje cifrado $c = m^n \bmod z$. Alicia manda c a Benito. Utilice el s secreto de Benito para descifrar el mensaje de Alicia: calcule $c^s \bmod z$ con *expomod*. ¿Es la respuesta igual a 123, tal como debía ser? En la vida real se utilizarían números mucho mayores.

Problema 7.33. Considere la matriz

$$F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Sean i y j dos enteros cualesquiera. ¿Cuál es el producto del vector (i, j) por la matriz F? ¿Qué ocurre si i y j son dos números consecutivos de la sucesión de Fibonacci? Utilice esta idea para desarrollar un algoritmo de divide y vencerás para calcular esta sucesión, y analice su eficiencia (1) considerando todas las operaciones aritméticas como de coste unitario y (2) contando con un tiempo que está en $\Theta(sq^{\alpha-1})$ para multiplicar enteros de tamaños q y s cuando $s \geq q$. Recuerde que el tamaño del n-ésimo número de Fibonacci está en $\Theta(n)$.

Problema 7.34. Representar el polinomio $p(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$, de grado d, mediante una matriz $P[0..d]$ que contenga sus coeficientes. Suponga que ya dispone de un algoritmo para multiplicar polinomios de grado k por polinomios de grado 1 en un tiempo que está en $O(k)$, así como otro algoritmo capaz de multiplicar dos polinomios de grado k en un tiempo que está en $O(k \log k)$. Sean los enteros $n_1, n_2, \ldots, n_d$. Desarrollar un algoritmo eficiente, basado en divide y vencerás, para hallar el único polinomio $p(n)$ de grado d cuyo coeficiente de mayor grado es 1, y tal que es $p(n_1) = p(n_2) = \ldots = p(n_d) = 0$. Analice la eficiencia de su algoritmo.

Problema 7.35. Sea $T[1..n]$ una matriz de n enteros. Se dice que un entero es un *elemento mayoritario* de T si aparece estrictamente más de $n/2$ veces en T. Dar un algoritmo que pueda decidir si una matriz $T[1..n]$ contiene un elemento mayoritario, y de ser así encontrarlo. El algoritmo debe de funcionar el tiempo lineal en el caso peor.

Problema 7.36. Rehacer el problema 7.35 con la limitación adicional consistente en que las únicas comparaciones admisibles entre los elementos de T tienen que ser comparaciones de igualdad. Por tanto, no se puede suponer que existe una relación de orden entre los elementos.

Problema 7.37. Si no ha podido hacer el problema 7.36, vuelva a intentarlo, pero permita que su algoritmo requiera un tiempo en $O(n \log n)$ en el caso peor.

Problema 7.38. Tiene usted que organizar un torneo con n participantes. Cada participante tiene que competir exactamente una vez con todos los posibles oponentes. Además, cada participante tiene que jugar exactamente un partido cada día, con la posible excepción de un solo día en el cual no juega.

(a) Si n es una potencia de 2, dar un algoritmo para construir un horario que permita que el torneo concluya en $n-1$ días.
(b) Para todo entero $n > 1$, dar un algoritmo para construir un horario que permita que finalice el torneo en $n-1$ días si n es par, o en n días si n es impar. Por ejemplo, la figura 7.7 da horarios posibles para torneos en que participan cinco y seis jugadores.

		Jugador				
	Día	1	2	3	4	5
	1	2	1	—	5	4
	2	3	5	1	—	2
($n = 5$)	3	4	3	2	1	—
	4	5	—	4	3	1
	5	—	4	5	2	3

		Jugador					
	Día	1	2	3	4	5	6
	1	2	1	6	5	4	3
	2	3	5	1	6	2	4
($n = 6$)	3	4	3	2	1	6	5
	4	5	6	4	3	1	2
	5	6	4	5	2	3	1

Figura 7.7. Horarios para cinco y seis jugadores

Problema 7.39. Se nos dan las coordenadas cartesianas de n puntos en el plano. Dar un algoritmo capaz de hallar el par de puntos más próximos en un tiempo que esté en $O(n \log n)$ en el caso peor.

Problema 7.40. Considere una matriz $T[1..n]$ y un entero k entre 1 y n. Utilice una simplificación para diseñar un algoritmo eficiente que intercambie los k primeros elementos de T con los $n-k$ últimos, sin hacer uso de una matriz auxiliar. Analice el tiempo de ejecución de su algoritmo.

Problema 7.41. Un *n-tally* es un circuito que admite n *bits* como entrada y produce $1 + \lfloor \lg n \rfloor$ *bits* como salida. Cuenta (en binario) el número de bits iguales a 1 que hay en la entrada. Por ejemplo, si $n = 9$ y las entradas son 011001011, la salida es 0101. Un *sumador-*(i, j) es un circuito que tiene una entrada de i bits, una entrada de j *bits*, y una salida de $[1 + \text{máx}(i, j)]$ *bits*. Suma sus dos entradas en binario. Por ejemplo, si $i = 3$, $j = 5$ y las entrada son 101 y 10111 respectivamente, la salida es 011100. Siempre es posible construir un *sumador-*(i,j) empleando exactamente $\max(i, j)$ circuitos 3-*tally*. Por esta razón, el 3-*tally* suele denominarse *sumador completo*.

(a) Utilizando sumadores completos y -(i, j) sumadores, mostrar la forma de construir un *n-tally* eficiente.

(b) Dar la recurrencia, incluidas las condiciones iniciales, para el número de 3-*tally* que se necesitan para construir el *n-tally* anterior. No olvide contar los 3-*tally* que forman parte de aquellos sumadores (i, j) que pueda utilizar.

(c) Utilizando la notación Θ, dar la expresión más sencilla posible para el número de 3-*tally* que se necesitan para construir el *n-tally* de los apartados anteriores. Justifique su respuesta.

Problema 7.42. Un *conmutador* es un circuito con dos entradas, un control y dos salidas. Conecta la entrada A con la salida A y la entrada B con la salida B, o bien la entrada A con la salida B y la entrada B con la salida A, dependiendo de la posición del control; véase la figura 7.8. Utilice estos conmutadores para construir una red con n entradas y n salidas, que sea capaz de implementar las $n!$ permutaciones posibles de las entradas. El número de conmutadores utilizado tiene que estar en $O(n \log n)$.

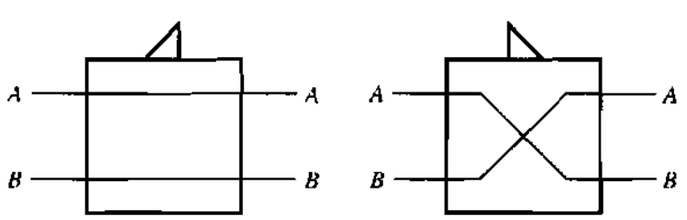


Figura 7.8. Conmutadores

7.10 REFERENCIAS Y TEXTOS MÁS AVANZADOS

El algoritmo para multiplicar enteros muy grandes en un tiempo que está en $O(n^{1.59})$ se atribuye a *Karatsuba* y *Ofman* (1962). Un algoritmo rápido para la multiplicación de enteros con hasta 10.000 cifras decimales se da en *Pollard* (1971). El algoritmo más rápido conocido, que puede multiplicar dos números de n cifras en un tiempo que está en $O(n \log n \log \log n)$, se debe a *Schönhage* y *Strassen* (1971); sus detalles se pormenorizan en *Brassard, Monet* y *Zuffellato* (1986). Una buena revisión de los algoritmos para la multiplicación de números enteros grandes se da

en la segunda edición (1981) de Knuth (1969), que incluye las respuestas de los problemas 7.2 y 7.3. Véase también Borodin y Munro (1975), y Turk (1982).

La técnica para determinar el umbral óptimo en el cual se debe de utilizar el subalgoritmo básico en lugar de dividir los subproblemas es original de Brassard y Bratley (1988). La solución del problema 7.11 también se da en Brassard y Bratley (1988); proporciona otra bonita aplicación de la técnica de inducción constructiva.

Quicksort procede de Hoare (1962). *Ordenar por fusión* y *quicksort* se describen con detalle en Knuth (1973), que es un compendio de técnicas de ordenación. El problema 7.14 fue resuelto por Kronrod; véase la solución del ejercicio 18 de la Sección 5.2.4 de Knuth (1973). El algoritmo lineal en el caso peor para la selección y búsqueda de la mediana se debe a Blum, Floyd, Pratt, Rivest y Tarjan (1972).

El algoritmo que multiplica dos matrices $n \times n$ en un tiempo que está en $O(n^{2.81})$ proviene de Strassen (1969). Los esfuerzos subsiguientes para mejorar el algoritmo de Strassen comenzaron con la demostración por Hopcroft y Kerr (1971) de que se necesitan siete multiplicaciones para multiplicar dos matrices 2×2 en una estructura no conmutativa; el primer éxito real fue obtenido por Pan (1980), y el algoritmo que es asintóticamente el más eficiente de los conocidos hasta el momento se debe a Coppersmith y Winograd (1990).

La idea de que se pueden lograr comunicaciones seguras a través de canales inseguros sin acuerdo previo acerca de algo secreto se debe independientemente a Merkle (1978) y a Diffie y Hellman (1976). El sistema criptográfico de clave pública RSA descrito en la Sección 7.8, que fue inventado por Rivest, Shamir y Adleman (1978) fue publicado por primera vez por Gardner (1977); tenga en cuenta, sin embargo, que en aquel desafío salieron victoriosos Atkins, Graff, Lenstra y Leyland en Abril de 1994, al cabo de ocho meses de cálculo en más de 600 computadoras de todo el mundo. El algoritmo eficiente capaz de violar este sistema en una computadora cuántica se debe a Shor (1994). Para más información acerca de la criptografía, consulte los artículos introductorios de Kahn (1966) y Hellman (1980), y los libros de Kahn (1967), Denning (1983), Kranakis (1986), Koblitz (1987), Brassard (1988), Simmons (1992), Schneier (1994) y Stimson (1995). Para un enfoque de criptografía que sigue siendo seguro independientemente de la potencia de cálculo del escucha, consulte Bennet, Brassard y Eckert (1992). La generalización natural del problema 7.28 se examina en Knuth (1969).

La solución del problema 7.33 se puede encontrar en Gries y Levin (1980) y en Urbanek (1980). El problema 7.39 está resuelto en Bentley (1980), pero consulte la Sección 10.9 para más información acerca de este problema. El problema 7.40 está resuelto en Gries (1981); véase también Brassard y Bratley (1988).

Capítulo



Programación dinámica

En el capítulo anterior veíamos que es posible dividir los ejemplares en subejemplares, resolver los subejemplares (posiblemente, subdividiéndolos de nuevo) y combinar entonces las soluciones de los subejemplares para resolver el caso original. A veces sucede que la forma natural de dividir un ejemplar, tal como lo sugiere la estructura del problema, nos lleva a considerar subejemplares solapados. Si los resolvemos todos independientemente, éstos darán a su vez lugar a toda una colección de subejemplares idénticos. Si no prestamos atención a esta duplicación, es probable que acabemos por tener un algoritmo ineficiente; si por el contrario aprovechamos la duplicación y nos las arreglamos para resolver cada subejemplar una sola vez, guardando la solución para su uso posterior, entonces tendremos un algoritmo más eficiente. La idea que subyace a la programación dinámica es, por tanto, bastante sencilla: evitar calcular dos veces una misma cosa, normalmente manteniendo una tabla de resultados conocidos que se vaya llenando a medida que se resuelven los subcasos.

El método de divide y vencerás es un método de *refinamiento progresivo*. Cuando se resuelve un problema mediante divide y vencerás, atacamos de inmediato el caso completo, que a continuación dividimos en subcasos más y más pequeños, a medida que progresa el algoritmo. La programación dinámica, por otra parte, es una técnica *ascendente*. Normalmente, se empieza por los subcasos más pequeños, y por tanto más sencillos. Combinando sus soluciones, obtenemos las respuestas para subcasos de tamaños cada vez mayores, hasta que finalmente llegamos a la solución del caso original.

Comenzaremos el capítulo por dos ejemplos sencillos de programación dinámica, que ilustran la técnica general en una situación poco complicada. Las secciones

siguientes abordan los problemas de devolver cambio, que mencionábamos en la Sección 6.1, y el de la mochila, que encontrábamos en la Sección 6.5.

8.1 DOS EJEMPLOS SENCILLOS

8.1.1 Cálculo del coeficiente binomial

Considere el problema consistente en calcular el coeficiente binomial

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{en caso contrario} \end{cases}$$

Supongamos que $0 \le k \le n$. Si calculamos directamente $\binom{n}{k}$ mediante

```
función C(n, k)
    si k = 0 o k = n entonces devolver 1
    sino devolver C(n–1, k–1) + C(n–1, k)
```

entonces muchos de los valores de $C(i, j)$, con $i < n$, $j < k$ se calculan una y otra vez. Por ejemplo, el algoritmo calcula $C(5,3)$ como la suma de $C(4,2)$ y $C(4,3)$. Ambos resultados intermedios exigen calcular $C(3,2)$. De forma similar, el valor de $C(2,2)$ se utiliza varias veces. Dado que el resultado final se obtiene sumando un cierto número de unos, el tiempo de ejecución de este algoritmo estará con certeza en $\Omega\binom{n}{k}$. Encontrábamos anteriormente un fenómeno similar, en el algoritmo *Fibrec* para calcular la sucesión de Fibonacci; véase la Sección 2.7.5.

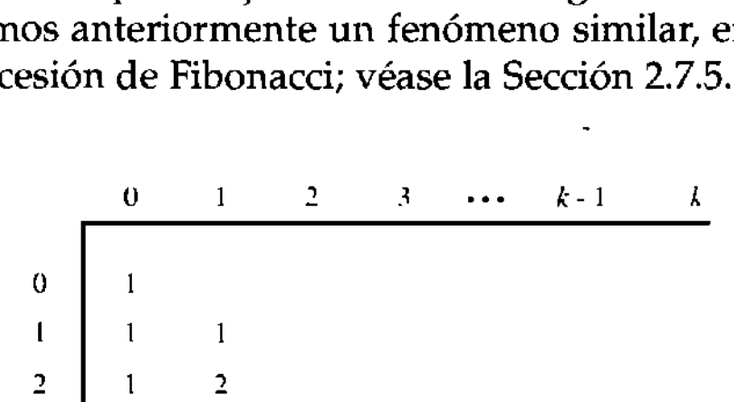


Figura 8.1. El triángulo de Pascal

Si por otra parte utilizamos una tabla de resultados intermedios —que por supuesto se trata del *triángulo de Pascal*—, entonces obtenemos un algoritmo más eficiente; véase la figura 8.1. La tabla debería ir llenándose línea por línea. De hecho, ni siquiera es necesario llenar toda la tabla: basta con mantener un vector k, que

representa la línea actual, y actualizar el vector de derecha a izquierda. Por tanto, para calcular $\binom{n}{k}$ el algoritmo requiere un tiempo que está en $\Theta(nk)$, y un espacio que está en $\Theta(k)$, si suponemos que la suma es una operación elemental.

8.1.2 El campeonato mundial

Considere una competición en la cual hay dos equipos A y B que juegan un máximo de $2n-1$ partidas, y en donde el ganador es el primer equipo que consiga n victorias. Suponemos que no hay empates, que los resultados de todos los partidos son independientes, y que para cualquier partido dado hay una probabilidad constante p de que el equipo A sea el ganador, y por tanto una probabilidad constante $q = 1 - p$ de que gane el equipo B.

Sea $P(i, j)$ la probabilidad de que el equipo A gane el campeonato, cuando todavía necesita i victorias más para conseguirlo, mientras que el equipo B necesita j victorias más para ganar. Por ejemplo, antes del primer partido del campeonato la probabilidad de que gane el equipo A es $P(n, n)$: ambos equipos necesitan todavía n victorias para ganar el campeonato. Si el equipo A necesita cero victorias más, entonces lo cierto es que ya ha ganado el campeonato, y por tanto $P(0, i) = 1$, con $1 \leq i \leq n$. De manera similar, si el equipo B necesita 0 victorias más, entonces ya ha ganado el campeonato, y por tanto $P(i, 0) = 0$, con $1 \leq i \leq n$. Como no puede producirse una situación en la que ambos equipos hayan ganado todos los partidos que necesitaban, $P(0, 0)$ carece de significado. Por último dado que el equipo A gana cualquier partido con una probabilidad p y pierde con una probabilidad q:

$$P(i, j) = pP(i-1, j) + qP(i, j-1), \quad i \geq 1, j \geq 1$$

Entonces podemos calcular $P(i, j)$ en la forma siguiente:

```
función P(i, j)
    si i = 0 entonces devolver 1
    sino si j = 0 entonces devolver 0
    sino devolver pP(i–1, j) + qP(i, j–1)
```

Sea $T(k)$ el tiempo necesario en el caso peor para calcular $P(i, j)$ en donde $k = i + j$. Con este método, vemos que es

$$T(1) = c$$
$$T(k) \leq 2T(k\text{-}1) + d, \quad k > 1$$

en donde c y d son constantes. Reescribiendo $T(k-1)$ en términos de $T(k-2)$, y así sucesivamente, obtenemos

$$
\begin{aligned}
T(k) &\le 4T(k-2) + 2d + d, \quad k > 1 \\
&\vdots \\
&\le 2^{k-1}T(1) + (2^{k-2} + 2^{k-3} + \ldots + 2 + 1)d \\
&= 2^{k-1}c + (2^{k-1} - 1)d \\
&= 2^k(c/2 + d/2) - d
\end{aligned}
$$

Por tanto $T(k)$ está en $O(2^k)$, que es $O(4^n)$ si $i = j = n$. De hecho, si examinamos la forma en que se generan las llamadas recursivas, encontramos el patrón que se muestra en la figura 8.2, que es idéntico al cálculo menos sofisticado del coeficiente binomial. Para ver esto imaginemos que toda llamada a $P(m, n)$ en la figura se sustituye por $C(m + n, n)$.

De esta manera, $P(i, j)$ se sustituye por $C(i + j, j)$,$P(i - 1, j)$ se sustituye por $C(i + j - 1, j)$ y $P(i, j-1)$ se sustituye por $C(i + j - 1, j-1)$. Ahora el patrón de llamadas que muestran las flechas corresponde al cálculo

$$C(i + j, j) = C(i + j - 1, j) + C(i + j - 1, j-1)$$

de un coeficiente binomial. El número total de llamadas recursivas, por tanto, es exactamente de $2\binom{i+j}{j}-2$; véase el Problema 8.1. Así pues para calcular la probabilidad $P(n, n)$ de que el equipo A sea el ganador, dado que todavía no ha empezado el campeonato, el tiempo requerido está, por tanto, en $\Omega\binom{2n}{n}$.

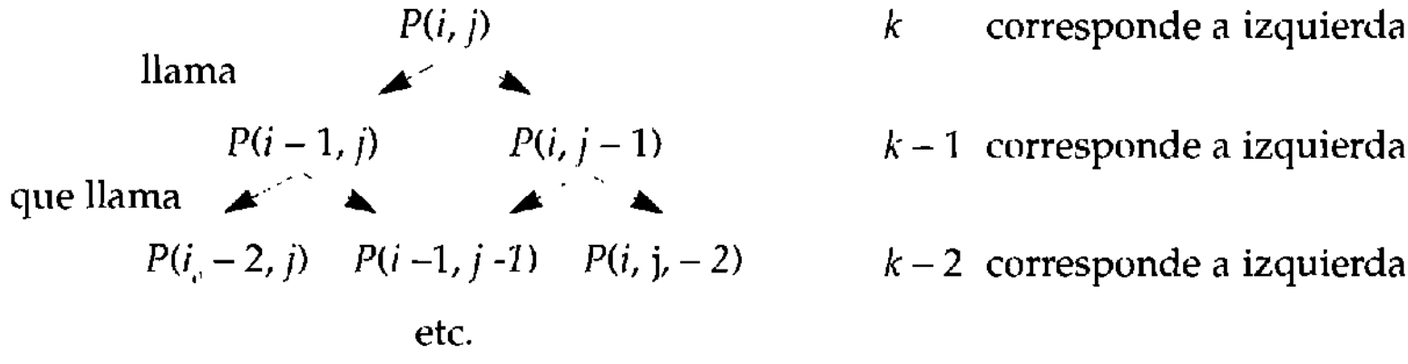


Figura 8.2. Llamadas recursivas efectuadas por una llamada a P(i, j)

El Problema 1.42 pide al lector demostrar que $\binom{2n}{n} \ge 4^n / (2n + 1)$. Combinando estos resultados, se ve que el tiempo necesario para calcular $P(n, n)$ está en $O(4^n)$ y en $\Omega(4^n / n)$. El método, por tanto, no resulta práctico para valores grandes de n. (Aun cuando las competiciones deportivas con $n > 4$ sean la excepción, ¡este problema tiene otras aplicaciones!)

Para acelerar el algoritmo, procederemos más o menos igual que con el triángulo de Pascal: declaramos un vector del tamaño adecuado y después vamos rellenando las entradas. Esta vez, sin embargo, en lugar de ir llenando el vector línea por línea, vamos a trabajar diagonal por diagonal. Éste es el algoritmo para calcular $P(n, n)$.

```
función serie(n, p)
    matriz P[0..n,0..n]
    q ← 1– p
    {Llenamos desde la esquina izquierda hasta la diagonal principal}
    para s ← 1 hasta n hacer
        P[0, s] ← 1; P[s, 0] ← 0
        para k ← 1 hasta s – 1 hacer
            P[k, s – k] ← pP[k–1, s – k] + qP[k, s – k – 1]
    {Llenamos desde debajo de la diagonal principal hasta la esquina derecha}
    para s ← 1 hasta n hacer
        para k ← 0 hasta n – s hacer
            P[s + k, n – k] ← pP[s + k – 1, n – k] + qP[s + k, n – k – 1]
    devolver P[n, n]
```

Dado que el algoritmo tiene que llenar una matriz $n \times n$, y dado que se necesita una constante temporal para calcular cada entrada, su tiempo de ejecución se encuentra en $O(n^2)$. Al igual que en el triángulo de Pascal, resulta sencillo implementar este algoritmo de tal manera que baste con un espacio de almacenamiento en $\Theta(n)$.

8.2 DEVOLVER CAMBIO (2)

Recuerde que el problema consiste en desarrollar un algoritmo para *pagar una cierta cantidad a un cliente,* empleando el menor número posible de monedas. En la Sección 6.1 describíamos un algoritmo voraz para este problema. Desafortunadamente, aunque el algoritmo voraz es muy eficiente, funciona solamente en un número limitado de casos. Con ciertos sistemas monetarios, o cuando faltan monedas de una cierta denominación (o su número es limitado), el algoritmo puede encontrar una respuesta que no sea óptima, o incluso puede no hallar respuesta ninguna.

Por ejemplo, supongamos que vivimos en un lugar en el cual hay monedas de 1, 4 y 6 unidades. Si tenemos que cambiar 8 unidades, el algoritmo voraz propondrá hacerlo con una moneda de 6 unidades y dos de una unidad, con un total de tres monedas. Sin embargo, está claro que podemos hacerlo mejor: basta con dar al cliente su cambio empleando tan sólo dos monedas de cuatro unidades. Aunque el algoritmo voraz no halla esta solución, resulta sencillo obtenerla empleando programación dinámica.

Como en la sección anterior, el *quid* del método consiste en preparar una tabla que contenga resultados intermedios útiles, que serán combinados en la solución del caso que estamos considerando. Supongamos que el sistema monetario que estamos considerando tiene monedas de n denominaciones diferentes, y que una moneda de denominación i, con $1 \leq i \leq n$ tiene un valor de d_i unidades. Supondremos, como es habitual, que todos los $d_i > 0$. Por el momento, supondremos también que se dispone de un suministro ilimitado de monedas de cada denominación. Por último, supongamos que tenemos que dar al cliente monedas por valor de N unidades, empleando el menor número posible de monedas.

Para resolver este problema mediante programación dinámica, preparamos una tabla $c[1..n, 0..N]$ con una fila para cada denominación posible y una columna para las cantidades que van desde 0 unidades hasta N unidades. En esta tabla, $c[i, j]$ será el número mínimo de monedas necesarias para pagar una cantidad de j unidades, con $0 \leq j \leq N$, empleando solamente monedas de las denominaciones desde 1 hasta i, con $1 \leq i \leq n$. La solución del ejemplar, por tanto, está dada por $c[n, N]$ si lo único que necesitamos saber es el número de monedas que se necesitan. Para rellenar la tabla, obsérvese primero que $c[i, 0]$ es cero para todos los valores de i. Después de esta inicialización, la tabla se puede rellenar o bien fila por fila de izquierda a derecha, o bien columna por columna avanzando hacia abajo. Para abonar una cantidad j utilizando monedas de las denominaciones entre 1 e i, tenemos dos opciones en general. En primer lugar, podemos decidir que no utilizaremos monedas de la denominación i, aun cuando esto está permitido ahora, en cuyo caso $c[i, j] = c[i-1, j]$. Como alternativa, podemos decidir que emplearemos al menos una moneda de la denominación i. En este caso, una vez que hayamos entregado la primera moneda de esta denominación, quedan por pagar $j - d_i$ unidades. Para pagar esto se necesitan $c[i, j - d_i]$ unidades, así que $c[i, j] = 1 + c[i, j - d_i]$. Dado que deseamos minimizar el número de monedas utilizadas, seleccionaremos aquella alternativa que sea mejor. En general, por tanto:

$$c[i, j] = \text{mín}(c[i-1, j], 1 + c[i, j - d_i])$$

Cuando $i = 1$, uno de los elementos que hay que comparar cae fuera de la tabla. Lo mismo sucede cuando $j < d_i$. Resulta cómodo pensar que tales elementos poseen el valor $+\infty$. Si $i = 1$ y $j < d_1$ entonces los dos elementos que hay que comparar caen fuera de la tabla. En este caso, hacemos $c[i, j]$ igual a $+\infty$ para indicar que es imposible pagar una cantidad j empleando solamente monedas del tipo 1.

La figura 8.3 ilustra el caso dado anteriormente, en el que teníamos que pagar 8 unidades con monedas que valían 1, 4 y 6 unidades. Por ejemplo, $c[3, 8]$ se obtiene en este caso como el menor de $c[2, 8] = 2$ y $1 + c[3, 8 - d_3] = 1 + c[3, 2] = 3$. Las entradas en el resto de la tabla se obtienen de forma similar. La respuesta para este caso concreto es que podemos pagar ocho unidades empleando únicamente dos monedas. De hecho, la tabla nos da la solución de nuestro problema para todos los casos que supongan un pago de 8 unidades o menos.

Cantidad:	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Figura 8.3 Devolver cambio empleando programación dinámica

Véase a continuación una versión más formal del algoritmo:

```
función monedas(N)
    {Devuelve el mínimo número de monedas necesarias
       para cambiar N unidades. El vector d[1..n]
       especifica las denominaciones: en el ejemplo hay monedas
       de 1, 4 y 6 unidades}
    vector d[1..n] = [1, 4, 6]
    matriz c[1..n, 0..N]
    para i ← 1 hasta n hacer c[i, 0] ← 0
    para i ← 1 hasta n hacer
       para j ← 1 hasta N hacer
             si i = 1 y j < d[i] entonces c[i, j] ← +∞
             sino si i = 1 entonces c[i, j] ← 1+ c[1, j – d[1]]
             sino si j < d[i] entonces c[i, j] ← c[i–1, j]
             sino c[i, j] ← mín(c[i – 1, j],1+c[i, j – d[i]])
    devolver c[n, N]
```

Si está disponible un suministro inagotable de monedas con un valor de una unidad, entonces siempre podemos hallar una solución para nuestro problema. De no ser así, puede haber valores de N para los cuales no exista una solución. Esto sucede, por ejemplo, si todas las monedas representan un número par de unidades, y se nos pide que paguemos un número impar de unidades. En tales casos, el algoritmo devuelve el resultado artificial $+\infty$. El problema 8.9 invita al lector a modificar el algoritmo para abordar una situación en que esté limitado el suministro de monedas de una cierta denominación.

Aun cuando el algoritmo sólo parece decir cuántas monedas se necesitan para dar el cambio correspondiente a una determinada cantidad, es fácil descubrir las monedas que se necesitan una vez que se ha construido la tabla c. Supongamos que es preciso abonar una cantidad j empleando monedas de las denominaciones 1, 2, ..., i. Entonces el valor de $c[i, j]$ dice cuántas monedas se van a necesitar. Si $c[i, j] = c[i - 1, j]$ entonces no se necesitan monedas de la denominación i, y pasamos a $c[i - 1, j]$, para ver lo que hay que hacer a continuación; si $c[i, j] = 1 + c[i, j - d_i]$ entonces entregamos una moneda de denominación i, que vale d_i y avanzamos hacia la izquierda hasta $c[i, j - d_i]$ para ver lo que hay que hacer a continuación. Si $c[i - 1, j]$ y $1 + c[i, j - d_i]$ son ambos iguales a $c[i, j]$ entonces podemos seleccionar cualquiera de las dos posibilidades. Siguiendo de esta manera, volveremos eventualmente a $c[0, 0]$ y ya no quedará nada por abonar. Esta fase del algoritmo es esencialmente un algoritmo voraz que basa sus decisiones en la información de la tabla, y no tiene que retroceder nunca.

El análisis del algoritmo es directo. Para ver cuántas monedas se necesitan para dar un cambio de N unidades cuando están disponibles monedas de n denominaciones diferentes, el algoritmo tiene que rellenar una matriz $n \times (N + 1)$, así que el tiempo de ejecución está en $\Theta(nN)$. Para ver las monedas que habría que utilizar, la búsqueda que retrocede desde $c[n, N]$ hasta $c[0, 0]$ da n–1 pasos hasta la fila de encima (que corresponde a no utilizar una moneda de la denominación ac-

tual) y $c[n, N]$ pasos hacia la izquierda (que corresponde a entregar una moneda). Dado que cada uno de estos pasos se puede dar en un tiempo constante, el tiempo total que se requiere está en $\Theta(n + c[n, N])$.

8.3 EL PRINCIPIO DE OPTIMALIDAD

La solución del problema de devolver cambio que se ha obtenido mediante programación dinámica parece sencilla, y no parece ocultar consideraciones teóricas. Sin embargo, es importante observar que se basa en un principio útil denominado *principio de optimalidad*, que en muchas circunstancias parece tan natural que se invoca casi sin pensarlo. Este principio afirma que en una sucesión óptima de decisiones u opciones, toda subsecuencia debe ser también óptima. En nuestro ejemplo, hemos dado por supuesto, cuando calculábamos $c[i, j]$ como el menor entre $c[i-1, j]$ y $1 + c[i, j-d_i]$, que si $c[i, j]$ es la forma óptima de devolver cambio de j unidades empleando monedas cuyas denominaciones estén entre 1 e i, entonces $c[i-1, j]$ y $c[i, j-d_i]$ deben dar también las soluciones *óptimas* de los casos que representan. En otras palabras, aun cuando el único valor de la tabla que nos interesa realmente es $c[n, N]$, hemos dado por supuesto que todas las demás entradas de la tabla deben de representar también las selecciones óptimas: y así es, porque en este problema es aplicable el principio de optimalidad.

Aun cuando este principio pueda parecer evidente, no es aplicable a todos los problemas que podamos encontrar. Cuando el principio de optimalidad *no* es aplicable, es probable que no sea posible atacar al problema en cuestión empleando programación dinámica. Tal es el caso, por ejemplo, de un problema que concierne a la utilización óptima de unos recursos limitados. Aquí la solución óptima de un caso puede no ser la obtenida por combinación de las soluciones óptimas de dos o más subcasos, si los recursos utilizados en esos subcasos suman más que el total de recursos disponibles.

Por ejemplo, si el camino más corto entre Montreal y Toronto pasa por Kingston, entonces la parte del camino que va desde Montreal hasta Kingston también debe de seguir el camino más corto posible, al igual que la parte del camino que une Kingston con Toronto. Por tanto, es aplicable el principio de optimalidad. Sin embargo, si el camino más rápido para ir desde Montreal hasta Toronto nos lleva a través de Kingston, no se sigue necesariamente que lo mejor sea ir tan rápidamente como sea posible desde Montreal hasta Kingston, y después ir tan rápidamente como sea posible desde Kingston hasta Toronto. Si utilizamos demasiada gasolina en la primera mitad del viaje, tendremos que repostar en algún sitio en la segunda parte, y así perderemos más tiempo que el ganado al conducir muy deprisa. Los subviajes desde Montreal hasta Kingston y desde Kingston hasta Toronto no son independientes, porque comparten un recurso, así que la selección de una solución óptima para un subviaje puede impedir que utilicemos una solución óptima para el otro. En esta situación, el principio de optimalidad no es aplicable.

Como segundo ejemplo, consideremos el problema consistente en hallar no el camino más corto, sino el camino simple más largo entre dos ciudades, empleando un conjunto dado de carreteras. Un *camino simple* es uno que nunca pasa dos veces

por el mismo sitio, así que esta condición elimina los caminos infinitos que dan vueltas y más vueltas en un bucle. Si sabemos que el camino simple más largo entre Montreal y Toronto pasa por Kingston, entonces *no* se sigue que sea posible obtenerlo tomando el camino simple más largo desde Montreal hasta Kingston, y después el camino simple más largo desde Kingston hasta Toronto. Es excesivo esperar que al pegar estos dos caminos simples se obtenga otro camino simple también. Una vez más, no es aplicable el principio de optimalidad.

Sin embargo, el principio de optimalidad resulta aplicable en muchas ocasiones. En tal caso, es posible volver a enunciarlo en la forma siguiente: la solución óptima de cualquier caso no trivial de un problema es una combinación de soluciones óptimas de *algunos* de sus subcasos. La dificultad para transformar este principio en un algoritmo es que no suele ser evidente cuáles son los subcasos relevantes para el caso considerado. Volviendo al ejemplo de encontrar el camino más corto, ¿cómo podemos saber si el subcaso que consiste en hallar el camino más corto entre Montreal y Ottawa es relevante cuando deseamos el camino más corto entre Montreal y Toronto? Esta dificultad nos impide utilizar una aproximación similar a divide y vencerás, comenzando en el caso original y buscando recursivamente soluciones óptimas para los subcasos relevantes, y sólo para estos. En su lugar, la programación dinámica resuelve todos los subcasos, para determinar los que realmente son relevantes; sólo entonces se combinan éstos en una solución óptima para el caso original.

8.4 EL PROBLEMA DE LA MOCHILA (2)

Como en la Sección 6.5, se nos da un cierto número de objetos y una mochila. Esta vez, sin embargo, suponemos que los objetos *no* se pueden fragmentar en trozos más pequeños, así que podemos decidir si tomamos un objeto o lo dejamos, pero no podemos tomar una fracción de un objeto. Para $i = 1, 2, \ldots, n$ supongamos que el objeto i tiene un peso positivo w_i y un valor positivo v_i. La mochila puede llevar un peso que no supere W. Nuestro objetivo es una vez más llenar la mochila de tal forma que se maximice el valor de los objetos incluidos, respetando la limitación de capacidad. Sea x_i igual a 0 si decidimos no tomar el objeto i, o bien 1 si incluimos el objeto i. En símbolos, el nuevo problema se puede enunciar en la forma siguiente:

$$\text{maximizar} \sum_{i=1}^{n} x_i v_i \qquad \text{con la restricción} \sum_{i=1}^{n} x_i w_i \leq W$$

donde $v_i > 0$, $w_i > 0$ y $x_i \in \{0, 1\}$ para $1 \leq i \leq n$. Aquí las condiciones que afectan a v_i y a w_i son limitaciones que afectan al caso; las de x_i son restricciones que afectan a la solución. Dado que el problema se parece mucho al de la Sección 6.5, es natural preguntarse en primer lugar si una ligera modificación del algoritmo voraz que utilizábamos antes podría funcionar ahora. Supongamos entonces que se adapta el algoritmo de la forma evidente, de tal modo que vaya examinando los objetos en orden descendente de valor por unidad de peso. Si la mochila no esta llena, el algoritmo tiene que seleccionar un objeto *completo* si es posible, antes de pasar al siguiente.

Desafortunadamente, el algoritmo voraz no funciona cuando x_i tiene que ser o bien 0 o bien 1. Por ejemplo, supongamos que están disponibles tres objetos, el primero de los cuales pesa 6 unidades y tiene un valor de 8, mientras que los otros dos pesan 5 unidades cada uno y tienen un valor de 5 cada uno. Si la mochila puede llevar 10 unidades, entonces la carga óptima incluye a los dos objetos más ligeros, con un valor total de 10. El algoritmo voraz, por otra parte, comenzaría por seleccionar el objeto que pesa 6 unidades, puesto que es el que tiene un mayor valor por unidad de peso. Sin embargo, si los objetos no se pueden romper, el algoritmo no podrá utilizar la capacidad restante de la mochila. La carga que produce, por tanto, consta de un solo objeto, y tiene un valor de 8 nada más.

Para resolver el problema por programación dinámica, preparamos una tabla $V[1..n, 0..W]$ que tiene una fila para cada objeto disponible, y una columna para cada peso desde 0 hasta W. En la tabla, $V[i, j]$ será el valor máximo de los objetos que podemos transportar si el límite de peso es j, con $0 \leq j \leq W$, y si solamente incluimos los objetos numerados desde el 1 hasta el i, con $1 \leq i \leq n$. La solución de este caso, por tanto, se puede encontrar en $V[n, W]$.

El paralelismo con el problema de dar la vuelta es bastante estricto. Una vez más, es aplicable el principio de optimalidad. Podemos rellenar la tabla bien fila por fila o bien columna por columna. En la situación general, $V[i, j]$ es el máximo (puesto que estamos intentando maximizar el valor) de $V[i-1, j]$ y $V[i-1, j-w_i]+v_i$. La primera de estas opciones corresponde a no añadir el objeto i a la carga. La segunda corresponde a seleccionar el objeto i, lo cual tiene como efecto incrementar el valor de la carga en v_i, y reducir en w_i la capacidad disponible. Por tanto, llenaremos las entradas de la tabla empleando la regla general

$$V[i, j] = \text{máx}(V[i-1, j], V[i-1, j-w_i]+v_i)$$

Para las entradas fuera de límites, definimos $V[0, j]$ igual a 0 cuando $j \geq 0$, y definimos $V[i, j]$ igual a $-\infty$ para todo i cuando $j < 0$. El enunciado formal del algoritmo, que se asemeja mucho a la función *monedas* de la sección anterior, se deja como ejercicio para el lector; véase el problema 8.11.

Límite de peso:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1, v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5, v_3=18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6, v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7, v_5=28$	0	1	6	7	7	18	22	28	29	34	35	40

Figura 8.4. La mochila, empleando programación dinámica

La figura 8.4 da un ejemplo del funcionamiento del algoritmo. En la figura hay cinco objetos, cuyos pesos son respectivamente 1, 2, 5, 6 y 7 unidades, y cuyos va-

lores son 1, 6, 18, 22 y 28. Sus valores por unidad de peso son por tanto 1,00, 3,00, 3,60, 3,67 y 4,00. Si sólo podemos transportar un máximo de 11 unidades de peso, entonces la tabla muestra que podemos componer una carga cuyo valor es 40.
De igual modo que para el problema de devolver cambio, la tabla V nos permite recuperar no sólo el valor de la carga óptima que podemos transportar, sino también su composición. En nuestro ejemplo, comenzamos por examinar $V[5, 11]$. Dado que $V[5, 11] = V[4, 11]$ pero $V[5, 11] \neq V[4, 11-w_5] + v_5$, una carga óptima no puede incluir al objeto 5. A continuación, $V[4, 11] \neq V[3, 11]$ pero $V[4, 11] = V[3, 11-w_4] + v_4$, así que una carga óptima debe incluir al objeto 4. Ahora $V[3, 5] \neq V[2, 5]$ pero $V[3, 5] = V[2, 5-w_3] + v_3$, así que debemos incluir el objeto 3. Prosiguiendo de esta forma, encontramos que $V[2, 0] = V[1,0]$ y que $V[1, 0] = V[0, 0]$, así que la carga óptima no incluye al objeto 2 ni al objeto 1. En este caso, por tanto, sólo hay una carga óptima, que consta de los objetos 3 y 4.

En este ejemplo, el algoritmo voraz consideraría primero el objeto 5, puesto que es el que tiene un mayor valor por unidad de peso. La mochila sólo puede llevar uno de estos objetos. A continuación, el algoritmo voraz consideraría el objeto 4, cuyo valor por unidad de peso es el siguiente. Este objeto no se puede incluir en la carga sin violar la restricción de capacidad. Prosiguiendo de esta manera, el algoritmo voraz examinaría los objetos 3, 2 y 1, por este orden, y acabaría por ofrecernos una carga que constaría de los objetos 5, 2 y 1 con un valor total de 35. Una vez más, vemos que el algoritmo voraz no funciona cuando no es posible romper los objetos.

El análisis del algoritmo de programación dinámica es directo, y se asemeja mucho al análisis del algoritmo para devolver cambio. Encontramos que se necesita un tiempo que está en $\Theta(nW)$ para construir la tabla V, y que la composición de la carga óptima se puede determinar en un tiempo que esta en $O(n + W)$.

8.5 CAMINOS MÍNIMOS

Sea $G = \langle N, A \rangle$ un grafo dirigido; N es el conjunto de nodos y A es el conjunto de aristas. Toda arista tiene asociada una longitud no negativa. Deseamos calcular la longitud del camino más corto entre cada par de nodo. Compare esto con la Sección 6.4, en la cual buscábamos la longitud de los caminos más cortos entre un nodo concreto, el nodo origen, y todos los demás nodos.

Como antes, supongamos que los nodos de G están numerados desde 1 hasta n, así que $N = \{1, 2, \ldots, n\}$, y supongamos que hay una matriz L que da las longitudes de las aristas, con $L[i, i] = 0$ para $i = 1,2,\ldots,n$, $L[i, j] \geq 0$ para todo i y j, y $L[i, j] = \infty$ si no existe la arista (i, j).

Es aplicable el principio de optimalidad: si k es un nodo del camino mínimo entre i y j, entonces la parte del camino que va desde i hasta k, y la parte del camino que va desde k hasta j debe ser óptimo también.

Construimos una matriz D que da la longitud del camino más corto entre un par de nodos. El algoritmo da a D el valor inicial L, esto es, las distancias directas

entre nodos. A continuación, efectúa n iteraciones. Después de la iteración k, D da la longitud de los caminos más cortos que utilizan solamente los nodos {1,2, ...k} como nodos intermedios. Al cabo de n iteraciones, D nos da por tanto la longitud de los caminos más cortos que utilicen alguno de los nodos de N como nodo intermedio, que es el resultado deseado. En la iteración k, el algoritmo debe comprobar para cada par de nodos (i, j) si existe o no un camino que vaya de i a j pasando por el nodo k, y que sea mejor que el camino óptimo actual que pasa sólo por los nodos {1, 2, ...k–1}. Si D_k representa la matriz D después de la k-ésima iteración (de modo que $D_0 = L$), entonces la comprobación necesaria puede implementarse en la forma

$$D_k[i, j] = \text{mín}(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$

donde utilizamos el principio de optimalidad para calcular el camino más corto que va desde i hasta j pasando por k. Tácitamente, hemos utilizado el hecho de que un camino óptimo que pase por k no visitará k dos veces.

$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

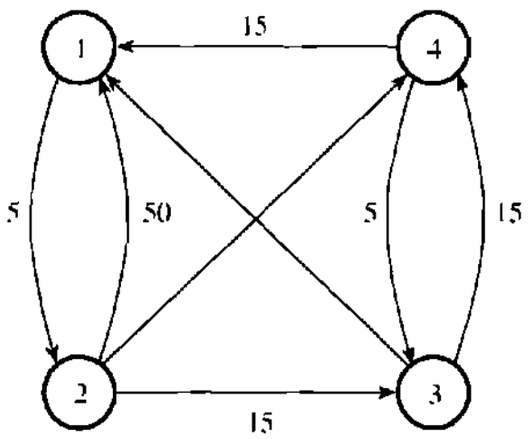


Figura 8.5. El algoritmo de Floyd en funcionamiento

En la k-ésima iteración, los valores de la k-ésima fila y de la k-ésima columna de D no cambian, porque $D[k, k]$ es siempre 0. Por tanto, no hace falta proteger estos valores al actualizar D. Esto nos permite utilizar una única matriz D n x n, mien-

tras que a primera vista pudiera parecer necesario utilizar dos de estas matrices, una que contendría los valores deD_{k-1} y otra los valores de D_k, o incluso una matriz $n \times n \times n$.

El algoritmo, que se conoce con el nombre de *algoritmo de Floyd*, es como sigue:

```
función Floyd(L[1..n, 1..n]): matriz[1..n, 1..n]
    matriz D[1..n, 1..n]
    D ← L
    para k ← 1 hasta n hacer
        para i ← 1 hasta n hacer
            para j ← 1 hasta n hacer
                D[i, j] ← mín(D[i, j], D[i, k] + D[k, j])
    devolver D
```

La figura 8.5 da un ejemplo de la forma en que funciona el algoritmo.

Es evidente que este algoritmo requiere un tiempo que está en $\Theta(n^3)$. También podemos utilizar el algoritmo de Dijkstra para resolver el mismo problema; véase la Sección 6.4. En este caso, tenemos que aplicar el algoritmo n veces, seleccionando en cada ocasión un nodo distinto como origen. Si utilizamos la versión del algoritmo de Dijkstra que funciona con una matriz de distancias, entonces el tiempo total de cálculo está en $n \times \Theta(n^2)$, esto es, en $\Theta(n^3)$. El orden es el mismo que para el algoritmo de Floyd, pero la sencillez de este último significa que es probable que tenga una constante oculta más pequeña, y por tanto que sea más rápido en la práctica. Además a los compiladores se les da bien optimizar los bucles **para**. Por otra parte, si utilizamos la versión del algoritmo de Dijkstra que emplea un montículo, y por tanto listas de las distancias hasta los nodos adyacentes, el tiempo total está en $n \times \Theta((a + n)\log n)$, esto es, en $\Theta((an + n^2)\log n)$, donde a es el número de aristas del grafo. Si el grafo es disperso ($a << n^2$), entonces quizá sea preferible utilizar el algoritmo de Dijkstra n veces; si el grafo es denso $a \approx n^2$, es mejor utilizar el algoritmo de Floyd.

Normalmente, deseamos saber cuál es el camino más corto, y no solamente su longitud. En tal caso, utilizaremos una segunda matriz P, cuyos elementos tienen todos ellos un valor inicial 0. El bucle más interno del algoritmo pasa a ser

```
si D[i, k]+D[k, j]<D[i, j] entonces  D[i, j] ← D[i, k]+D[k, j]
                                     P[i, j] ← k
```

Cuando se detiene el algoritmo, $P[i, j]$ contiene el número de la última iteración que haya dado lugar a un cambio en $D[i, j]$. Para recuperar el camino más corto desde i hasta j, se examina $P[i, j]$. Si $P[i, j] = 0$, entonces $D[i, j]$ nunca ha cambiado, y el camino mínimo pasa directamente por la arista (i, j); en caso contrario, si $P[i, j] = k$, entonces el camino más corto desde i hasta j pasa por k. Se examina recursivamente $P[i, k]$ y $P[k, j]$ para buscar cualquier otro posible nodo intermedio que esté en el camino más corto.

Si tomamos como ejemplo el grafo de la figura 8.5, entonces P pasa a ser

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Dado que $P[1, 3] = 4$, el camino mínimo desde 1 hasta 3 pasa por 4. Examinando ahora $P[1, 4]$ y $P[4, 3]$, descubrimos que entre 1 y 4 hay que pasar por 2, pero que de 4 a 3 pasamos directamente. Por último, vemos que los recorridos desde 1 hasta 2 y desde 2 hasta 4 también son directos. Por tanto, el camino mínimo desde 1 hasta 3 es 1, 2, 4, 3.

Si permitimos que las aristas del grafo tengan longitudes negativas, entonces la noción de «camino más corto» pierde gran parte de su significado: si el grafo incluye un ciclo cuya longitud total sea negativa, ¡cuanto más pasemos por el bucle, más corto será nuestro camino! El Problema 8.17 pregunta lo que sucede con el algoritmo de Floyd si le damos un grafo con aristas negativas, pero sin ciclos negativos. Aun cuando un grafo tenga ciclos negativos, sigue teniendo sentido pedir los caminos simples más cortos. (Recuerde que un camino simple es aquella que nunca visita dos veces el mismo nodo.) No se conoce un algoritmo eficiente para hallar los caminos simples más cortos en grafos que puedan tener aristas de longitud negativa. La situación es la misma que para el problema de hallar los caminos simples más largos, mencionado en la Sección 8.3: no se conoce un algoritmo eficiente. Estos dos problemas son *NP*-completos; véase el Capítulo 12.

8.6 MULTIPLICACIÓN ENCADENADA DE MATRICES

Recuerde que el producto de una matriz $p \times q$ llamada A por una matriz $q \times r$ llamada B es la matriz $p \times r$ dada por

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \qquad 1 \le i \le p, 1 \le j \le r$$

Algorítmicamente, se puede expresar esto en la forma

```
para i ← 1 hasta p hacer
   para j ← 1 hasta r hacer
      C[i, j] ← 0
      para k ← 1 hasta q hacer
         C[i, j] ← C[i, j] + A[i, k]B[k, j]
```

de donde se deduce claramente que se necesitan pqr multiplicaciones escalares para calcular la matriz producto empleando este algoritmo. (En esta sección no consideramos la posibilidad de utilizar un algoritmo mejor para la multiplicación de matrices, como puede ser el algoritmo de Strassen, descrito en la Sección 7.6.)

Supongamos ahora que deseamos calcular el producto de más de dos matrices. La multiplicación de matrices es asociativa, así que podemos calcular el producto matricial

$$M = M_1 M_2 \cdots M_n$$

de muchas maneras, todas las cuales darán la misma respuesta:

$$\begin{aligned} M &= (\cdots((M_1 M_2)M_3)\cdots M_n) \\ &= (M_1(M_2(M_3 \cdots (M_{n-1}M_n)\cdots))) \\ &= (\cdots((M_1 M_2)(M_3 M_4))\cdots), \end{aligned}$$

y así sucesivamente. Sin embargo, la multiplicación de matrices no es conmutativa, así que no se nos permite modificar el orden de las matrices.

La selección de método de cálculo puede tener una notable influencia sobre el tiempo requerido. Supongamos, por ejemplo, que deseamos calcular el producto $ABCD$ de cuatro matrices, en donde A es 13 × 5, B es 5 × 89, C es 89 × 3 y D es 3 × 34. Para medir la eficiencia de los diferentes métodos, contaremos el número de multiplicaciones escalares implicadas. Tal como se ha programado anteriormente, habrá un número igual de sumas escalares, además de algo de mantenimiento, así que el número de multiplicaciones escalares es un buen indicador de la eficiencia general. Por ejemplo, utilizando $M = ((AB)C)D$, se calculan sucesivamente

AB	5.785 multiplicaciones
$(AB)C$	3.471 multiplicaciones
$((AB)C)D$	1.324 multiplicaciones

con un total de 10.582 multiplicaciones escalares. Hay cinco formas esencialmente diferentes de calcular el producto en este caso: cuando se expresa el producto en la forma $(AB)(CD)$, no distinguimos entre el método que calcula primero AB y después CD y el que empieza por CD y después calcula AB, puesto que ambos requieren el mismo número de multiplicaciones. Para cada uno de estos cinco métodos, los números correspondientes de multiplicaciones escalares son:

$((AB)C)D$	10.582
$(AB)(CD)$	54.201
$(A(BC))D$	2.856
$A((BC)D)$	4.055
$A(B(CD))$	26.418

El método más eficiente es casi 19 veces más rápido que el más lento.

Para hallar directamente la mejor manera de calcular el producto, podríamos limitarnos a poner paréntesis en la expresión de todas las maneras posibles, contando en cada ocasión el número de multiplicaciones escalares que se necesita. Sea $T(n)$ el número de formas esencialmente distintas de poner paréntesis en un producto de n matrices. Supongamos que decidimos hacer el primer corte entre las matrices i-ésima e $(i+1)$-ésima del producto, en la forma siguiente:

$$(M_1 M_2 \cdots M_i)(M_{i+1} M_{i+2} \cdots M_n)$$

Ahora hay $T(i)$ formas de poner paréntesis en el término del lado izquierdo, y $T(n-i)$ formas de poner paréntesis en el termino del lado derecho. Cualquier forma de las primeras se puede combinar con cualquiera de las segundas, así que para este valor concreto de i hay $T(i)T(n-i)$ formas de poner paréntesis en toda la expresión. Dado que i puede tomar cualquier valor entre 1 y $n-1$, obtenemos finalmente la recurrencia siguiente para $T(n)$:

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

Añadiendo la condición inicial evidente, $T(1) = 1$, podemos utilizar la recurrencia para calcular cualquier valor que se necesite de T. La tabla siguiente muestra algunos de los valores de $T(n)$.

n	1	2	3	4	5	10	15
$T(n)$	1	1	2	5	14	4.862	2.674.440

Los valores de $T(n)$ se llaman *números de Catalan*.

Para cada manera de poner paréntesis en la expresión de M, se necesita un tiempo que está en $\Omega(n)$ para contar el número de multiplicaciones escalares que se necesitan (al menos, si no intentamos ser sutiles). Dado que $T(n)$ está en $\Omega(4^n / n^2)$ (combine los resultados de los problemas 8.24 y 1.42), la búsqueda de la mejor manera de calcular M empleando la aproximación directa requiere un tiempo que está en $\Omega(4^n / n)$. Este método, por tanto, no resulta práctico para valores grandes de n: hay demasiadas maneras de insertar paréntesis para examinarlas todas.

Un poco de experimentación mostrará que ninguno de los algoritmos voraces evidentes es capaz de calcular productos matriciales de forma óptima; véase el Problema 8.20. Afortunadamente, el principio de optimalidad es aplicable este problema. Por ejemplo, si la mejor manera de multiplicar todas las matrices nos exige hacer el primer corte entre las matrices i-ésima e $(i+1)$-ésima del producto, entonces los dos subproductos $M_1 M_2 \ldots M_i$ y $M_{i+1} M_{i+2} \ldots M_n$ también deben calcularse de forma óptima. Esto sugiere que deberíamos considerar la posibilidad de utilizar la programación dinámica. Construimos una tabla m_{ij}, $1 \le i \le j \le n$, en donde m_{ij} da la solución óptima, esto es, el número requerido de multiplicaciones escalares para la parte $M_i M_{i+1} \ldots M_j$ del producto solicitado. La solución del problema original, por tanto, viene dada por m_{1n}.

Supongamos que las dimensiones de las matrices están dadas por un vector $d[0..n]$ tal que la matriz M_i, con $1 \le i \le n$, es de dimensión $d_{i-1} \times d_i$. Construimos la tabla m_{ij} diagonal por diagonal: la diagonal s contiene los elementos m_{ij} tales que $j - i = s$. Por tanto la diagonal $s = 0$ contiene los elementos m_{ii} con $1 \le i \le n$, correspondientes a los «productos» M_i. Aquí no hay que hacer ninguna multiplicación, así que $m_{ii} = 0$ para todo i. La diagonal $s = 1$ contiene los elementos $m_{i,i+1}$ correspondientes a productos de la forma $M_i M_{i+1}$. Aquí no tenemos más opción que calcular directamente el producto, lo cual se puede hacer efectuando $d_{i-1}d_i d_{i+1}$ multiplicaciones escalares, tal como veíamos al principio de la sección. Finalmente, cuando sea $s > 1$, la diagonal s contendrá los elementos $m_{i,i+s}$ correspondientes a productos de la forma $M_i M_{i+1} \ldots M_{i+s}$. Ahora tenemos una opción: podemos hacer el primer corte en el producto después de cualquiera de las matrices $M_i M_{i+1}, \ldots M_{i+s-1}$. Si hacemos el corte después de M_k, con $i \le k < i + s$, entonces se necesitan m_{ik} multiplicaciones escalares para calcular el término de la izquierda, $m_{k+1,\,i+s}$ para calcular el término de la derecha, y por último $d_{i-1}d_k d_{i+s}$ para multiplicar las dos matrices resultantes con objeto de obtener el resultado final. Para hallar el óptimo, seleccionamos aquel corte que minimice el número requerido de multiplicaciones escalares.

En resumen, llenamos la tabla m_{ij} utilizando las reglas siguientes para $s = 0, 1, 2, \ldots, n-1$:

$$
\begin{array}{rlll}
s = 0: & m_{ii} & = 0 & i = 1, 2, \ldots, n \\
s = 1: & m_{i,i+1} & = d_{i-1}d_i d_{i+1} & i = 1, 2, \ldots, n-1 \\
1 < s < n: & m_{i,i+s} & = \min_{i \le k < i+s} (m_{ik} + m_{k+1,\,i+s} + d_{i-1}d_k d_{i+s}) & i = 1, 2, \ldots, n-s
\end{array}
$$

Sólo se necesita escribir explícitamente el segundo caso por claridad, puesto que cae dentro del caso general con $s = 1$.

Para aplicar esto al ejemplo, deseamos calcular el producto $ABCD$ de cuatro matrices, donde A es 13×5, B es 5×89, C es 89×3 y D es 3×34. El vector d es por tanto (13, 5, 89, 3, 34). Para $s = 1$, encontramos que es $m_{12} = 5785$, $m_{23} = 1335$ y $m_{34} = 9078$. A continuación, para $s = 2$ obtenemos

$$
\begin{aligned}
m_{13} &= \min (m_{11} + m_{23} + 13 \times 5 \times 3,\ m_{12} + m_{33} + 13 \times 89 \times 3) \\
&= \min (1.530,\ 9.256) = 1.530 \\
m_{24} &= \min (m_{22} + m_{34} + 5 \times 89 \times 34,\ m_{23} + m_{44} + 5 \times 3 \times 34) \\
&= \min (24.208,\ 1.845) = 1.845
\end{aligned}
$$

Finalmente, para $s = 3$

$$
\begin{aligned}
m_{14} = \min (&m_{11} + m_{24} + 13 \times 5 \times 34, && \{k = 1\} \\
&m_{12} + m_{34} + 13 \times 89 \times 34, && \{k = 2\} \\
&m_{13} + m_{44} + 13 \times 3 \times 34, && \{k = 3\} \\
= \min &(4.055,\ 54.201,\ 2.856) = 2.856
\end{aligned}
$$

La matriz completa m se muestra en la figura 8.6.

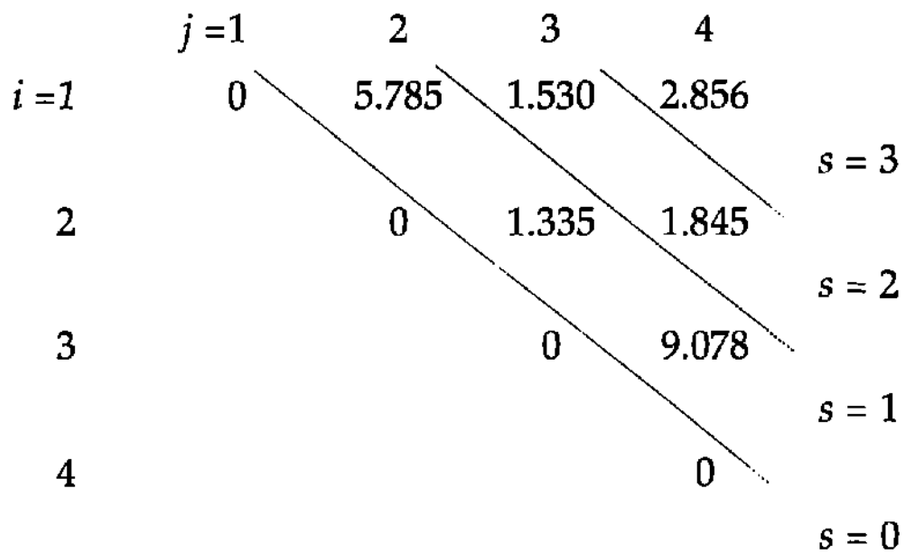


Figura 8.6. Un ejemplo del algoritmo de multiplicación de matrices encadenadas

Una vez más, lo normal será que no sólo queramos saber el número de multiplicaciones escalares necesarias para calcular el producto M, sino también la forma de efectuar eficientemente este cálculo. Como en la Sección 8.5, hacemos esto añadiendo una segunda matriz para llevar la cuenta de las opciones que vayamos tomando. Sea *mejork* esta nueva matriz. Ahora, cuando calculemos m_{ij} guardaremos en *mejork*[i, j] el valor de k que corresponde al término mínimo entre los comparados. Cuando se detenga el algoritmo, *mejork*[1, n] nos dirá dónde hay que hacer el primer corte del producto. Procediendo de forma recursiva en los dos términos producidos de esta manera, podemos reconstruir la forma óptima de poner paréntesis en M. Los problemas 8.21 y 8.22 le invitan a rellenar los detalles.

Para $s > 0$ hay que calcular $n - s$ elementos en la diagonal s; para cada uno de ellos, necesitamos decidirnos entre s posibilidades dadas por los distintos valores de k. Por tanto el tiempo de ejecución del algoritmo está en el orden exacto de

$$\sum_{s=1}^{n}(n-s)s = n\sum_{s=1}^{n-1}s - \sum_{s=1}^{n-1}s^2$$

$$= n^2(n-1)/2 - n(n-1)(2n-1)/6$$

$$= (n^3 - n)/6$$

donde hemos utilizado las proposiciones 1.7.14 y 1.7.15 para evaluar las sumas. El tiempo de ejecución de este algoritmo está entonces en $\Theta(n^3)$, si bien existen mejores algoritmos.

8.7 ENFOQUES QUE APLICAN RECURSIÓN

Aun cuando los algoritmos de programación dinámica, tal como el que se acaba de dar para calcular m, son eficientes, hay algo poco satisfactorio acerca del enfoque ascendente. Un método descendente, bien sea divide y vencerás, refinamiento progresivo o recursión, parece más natural, sobre todo para aquellos a quienes se ha enseñado a desarrollar programas de esta manera. Una razón de más peso es que el enfoque ascendente nos lleva a calcular valores que podrían ser totalmente irrelevantes. Resulta tentador, por tanto, ver si podemos alcanzar la misma eficiencia en una versión descendente del algoritmo.

Ilustramos esta idea con el problema de multiplicación matricial descrito en la sección anterior. Una línea de ataque sencilla consiste en sustituir la tabla m por una función fm, que se calcula cuando es necesario. En otras palabras, desearíamos encontrar una función fm tal que $fm(i, j) = m_{ij}$ para $1 \le i \le j \le n$, pero que pueda ser calculada recursivamente, a diferencia de la tabla m, que calculábamos de forma ascendente.

Resulta sencillo escribir esta función, lo único que hay que hacer es programar las reglas para calcular m:

```
función fm(i, j)
    si i = j entonces {sólo hay una matriz }
        devolver 0
    m ← ∞
    para k ← i hasta j–1 hacer
            m ← mín(m, fm(i, k) + fm(k+1, j)+d[i–1]d[k]d[j])
    devolver m
```

El vector global $d[0..n]$ da las dimensiones de las matrices que hay que multiplicar, exactamente igual que antes. Para todos los valores relevantes de k los intervalos $[i..k]$ y $[k+1..j]$ implicados en las llamadas recursivas afectan a menos matrices que $[i..j]$. Sin embargo, toda llamada recursiva sigue afectando al menos a una matriz (siempre y cuando, por supuesto, $i \le j$ en la llamada original). Eventualmente, por tanto, la recursión finaliza. Para averiguar cuántas multiplicaciones escalares se necesitan para calcular $M = M_1 M_2 \cdots M_n$, nos limitamos a llamar a $fm(1, n)$.

Para analizar este algoritmo, sea $T(s)$ el tiempo requerido para ejecutar una llamada a $fm(i, i + s)$, donde s es el número de multiplicaciones matriciales en el producto correspondiente. Éste es el mismo s que se utilizaba anteriormente para numerar las diagonales de la tabla m. Es claro que $T(0) = c$ para alguna constante c. Cuando $s > 0$, tenemos que seleccionar el menor entre s términos, cada uno de los cuales es de la forma

$$fm(i, k) + fm(k+1, i + s) + d[i–1]d[k]d[i + s], \quad i \le k < i + s$$

Sea d una constante tal que se pueden ejecutar dos multiplicaciones escalares, seis adiciones escalares, una comparación con el valor anterior del mínimo y el mantenimiento necesario, todo ello en un tiempo d. Ahora el tiempo necesario para evaluar uno de estos términos es $T(k-i) + T(i+s-k-1) + d$. El tiempo total $T(s)$ necesario para evaluar $fm(i, i+s)$ es por tanto

$$T(s) = \sum_{k=i}^{i+s-1} (T(k-i) + T(i+s-k-1) + b).$$

Si hacemos $m = k - i$, entonces esta expresión se convierte en

$$T(s) = \sum_{m=0}^{s-1} (T(m) + T(s-1-m) + b)$$

$$= sb + 2\sum_{m=0}^{s-1} T(m).$$

Dado que esto implica que $T(s) \geq 2T(s\text{-}1)$, vemos inmediatamente que $T(s) \geq 2^s T(0)$, así que el algoritmo requiere ciertamente un tiempo en $\Omega(2^n)$ para encontrar la mejor manera de multiplicar n matrices. Por tanto, no puede ser competitivo con el algoritmo que utiliza programación dinámica y que requiere un tiempo que está en $\Theta(n^3)$.

El algoritmo se puede acelerar si somos inteligentes y evitamos las llamadas recursivas cuando $d[i-1]d[k]d[i+s]$ ya es mayor que el valor anterior del mínimo. La mejora dependerá del caso, pero es muy improbable que un algoritmo que requiere un tiempo exponencial sea competitivo con respecto a uno que sólo requiere un tiempo polinómico.

Para encontrar una cota superior del tiempo requerido por el algoritmo recursivo, utilizamos inducción constructiva. Examinando la forma de la recursión para T, y recordando que $T(s) \geq 2^s T(0)$, parece posible que T estuviera acotado por una potencia de alguna constante mayor que 2: intentemos demostrar que $T(s) \leq a3^s$, para alguna constante adecuada a. Tomemos esto como hipótesis de inducción, y supongamos que es cierto para todo $m < s$. Al sustituir en la recurrencia, obtenemos

$$T(s) \leq sd + 2\sum_{m=0}^{s-1} a3^m$$

$$= sd + a3^s - a$$

donde hemos usado la Proposición 1.7.10 para calcular la suma. Desafortunadamente de aquí no podemos concluir que $T(S) \leq a\,3^1 - a$. Sin embargo, si adoptamos la táctica recomendada en la Sección 1.6.4 y reforzamos la hipótesis de inducción, las cosas salen mejor. Como hipótesis de inducción reforzada, suponemos que $T(m) \leq a\,3^m - b$ para $m < s$ donde b no es una constante desconocida. Ahora, al sustituir en la recurrencia obtenemos

$$T(s) \leq sd + 2\sum_{m=0}^{s-1}(a3^m - b)$$
$$= s(d - 2b) + a3^s - a$$

Esto es suficiente para asegurar que $T(s) \leq a3^s - b$, siempre y cuando $b \geq d/2$ y $a \geq b$. Para empezar la inducción, se necesita que $T(0) \leq a - b$, lo cual se satisface siempre y cuando $a \geq T(0) + b$. En resumen, hemos demostrado que $T(s) \leq a3^m - b$ para todo s, siempre y cuando $b \geq d/2$ y $a \geq T(0) + b$. El tiempo requerido por el algoritmo recursivo para calcular la mejor forma de realizar un producto de n matrices está por tanto en $O(3^n)$.

Concluimos que una llamada a la función recursiva *fm*(1,*n*) es más rápida que probar inocentemente todas las posibles formas de poner paréntesis en el producto deseado, lo cual, como ya se vio, requiere un tiempo que está en $\Omega(4^n / n)$. Sin embargo, es más lento que el algoritmo de programación dinámica descrito anteriormente. Esto ilustra algo que se decía anteriormente en este capítulo. Para decidir la mejor manera de poner paréntesis en el producto *ABCDEFG*, digamos, *fm* resuelve recursivamente 12 subcasos, incluyendo *ABCDEF* y *BCDEFG*, que se solapan, y resuelven ambos recursivamente *BCDEF* partiendo de cero. Esta duplicación de esfuerzos es lo que hace ineficiente a *fm*.

8.8 FUNCIONES CON MEMORIA

El algoritmo de la sección anterior no es el primero que hemos visto en el que una formulación recursiva sencilla de una solución da lugar a un programa ineficiente, porque hay subcasos comunes que se resuelven independientemente más de una vez. La programación dinámica nos permite evitar esta repetición, a costa de complicar el algoritmo. En la programación dinámica, además, es posible que tengamos que resolver algunos subcasos irrelevantes, puesto que sólo posteriormente conoceremos con exactitud las subsoluciones que se necesitan. Un algoritmo descendente recursivo no tiene esta limitación. ¿Será posible combinar las ventajas de ambas técnicas, y mantener la sencillez de una formulación recursiva sin perder la eficiencia que ofrece la programación dinámica?

Una forma sencilla de hacer esto que funciona en muchas situaciones, consiste en utilizar una *función con memoria*. Al programa recursivo se le añade una tabla del tamaño necesario. Inicialmente, todas las entradas de esta tabla contienen un valor especial que muestra que todavía no se han calculado. En lo sucesivo, cada vez que se llama a la función, se examina primero la tabla para ver si ya ha sido evaluada con el mismo conjunto de parámetros. De ser así, se devuelve el valor de la tabla. En caso contrario, seguimos adelante y calculamos la función. Antes de devolver el valor calculado, sin embargo, lo almacenamos en el lugar adecuado de la tabla. De esta forma nunca es necesario calcular dos veces la función para los mismos valores de sus parámetros.

Para el algoritmo recursivo *fm* de la Sección 8.7, sea *mtab* una tabla cuyas entradas se inicializan todas a –1 (puesto que el número de multiplicaciones

escalares necesarias para calcular un producto de matrices no puede ser negativo). La siguiente reformulación de la función *fm*, que utiliza la tabla *mtab* como variable global, combina la claridad de una formulación recursiva con la eficiencia de la programación dinámica:

```
función fm-mem(i, j)
    si i = j entonces devolver 0
    si mtab[i, j] ≥ 0 entonces devolver mtab[i, j]
    m ← ∞
    para k ← i hasta j–1 hacer
        m ← mín(m, fm-mem(i, k) + fm-mem(k + 1, j)
                                + d[i – 1]d[k]d[j])
    mtab[i, j] ← m
    proporcionar m
```

Como se indicaba en la Sección 8.7, esta función se puede acelerar evitando las llamadas recursivas si $d[i-1]d[k]d[j]$ ya es mayor que el valor anterior de m.

En algunas ocasiones, el uso de esta técnica exige pagar un cierto precio. Por ejemplo, en la Sección 8.1.1 vimos que se podía calcular el coeficiente binomial $\binom{n}{k}$ utilizando un tiempo que estaba en $\Theta(nk)$ y un espacio que estaba en $\Theta(k)$. Cuando se implementa utilizando una función con memoria, el cálculo requiere la misma cantidad de tiempo, pero necesita un espacio que está en $\Omega(nk)$, véase el Problema 8.26.

Si utilizamos un poco más de espacio (el espacio necesario sólo se multiplica por un factor constante), entonces podemos evitar el tiempo de inicialización que se necesita para dar un valor especial a todas las entradas de la tabla. Se puede hacer esto empleando una *inicialización virtual*, que se describía en la Sección 5.1. Esto resulta especialmente deseable cuando sólo es preciso calcular unos pocos valores de la función, pero no sabemos cuáles por anticipado. Por ejemplo, véase la Sección 9.1.

8.9 PROBLEMAS

Problema 8.1. Demostrar que el número total de llamadas recursivas efectuadas durante el cálculo de $C(n, k)$ empleando el algoritmo de la Sección 8.1.1 es exactamente $2\binom{n}{k} - 2$.

Problema 8.2. El cálculo de la sucesión de Fibonacci ofrece otro ejemplo de la clase de técnica presentada en la Sección 8.1. ¿Qué algoritmo de la Sección 2.7.5 utiliza programación dinámica?

Problema 8.3. Demostrar que el tiempo necesario para calcular $P(n, n)$ empleando la función P de la Sección 8.1.2 está en $O\left(4^n / \sqrt{n}\right)$.

Problema 8.4. Utilizando el algoritmo *campeonato* de la Sección 8.1.2, calcular la probabilidad de que el equipo A gane el campeonato si $p = 0{,}45$ y si se necesitan cuatro victorias para ganar.

Problema 8.5. Repetir el problema anterior con $p = 0{,}55$. ¿Cuál debería ser la relación entre las respuestas a estos dos problemas?

Problema 8.6. Igual que en el problema 8.4, calcular la probabilidad de que el equipo A gane el campeonato si $p = 0{,}45$ y se necesitan cuatro victorias para ganar. Esta vez, sin embargo, calcular la probabilidad requerida directamente como la probabilidad de que el equipo A gane 4 o más partidos de una serie de 7. (Jugar partidos adicionales después de que el equipo A haya ganado el campeonato no puede modificar el resultado.)

Problema 8.7. Adaptar el algoritmo *campeonato* de la Sección 8.1.2 al caso en que el equipo A gane cualquier partido dado con una probabilidad p, y que lo pierda con una probabilidad q, pero habiendo también una probabilidad r de que haya empate, lo que no supone una victoria para ningún equipo. Suponga que siguen siendo necesarias n victorias para ganar el campeonato. Desde luego, tiene que ser $p + q + r = 1$.

Problema 8.8. Demostrar que un espacio de almacenamiento en $\Theta(n)$ basta para implementar el algoritmo *campeonato* de la Sección 8.1.2.

Problema 8.9. Adaptar el algoritmo *monedas* de la Sección 8.2 para que funcione correctamente aun cuando el número de monedas de una cierta denominación esté limitado.

Problema 8.10. Rehacer el ejemplo ilustrado en la figura 8.4, pero renumerando los objetos por orden inverso (de tal modo que $w_1 = 7$, $v_1 = 28$,...,$w_5 = 1$, $v_5 = 1$). ¿Qué elementos de la tabla no deben resultar afectados?

Problema 8.11. Escribir el algoritmo para rellenar la tabla V según se describe en la Sección 8.4.

Problema 8.12. Cuando $j < w_i$ en el algoritmo para rellenar la tabla V, según se describe en la Sección 8.4, consideramos que $V[i-1, j - w_i]$ es $-\infty$. ¿Puede haber entradas que valgan $-\infty$ en la tabla ya terminada? De ser así, ¿qué indican? En caso contrario, ¿por qué no?

Problema 8.13. Puede haber más de una solución óptima para un caso del problema de la mochila. Empleando la tabla V descrita en la Sección 8.4, ¿es posible encontrar todas las posibles soluciones óptimas de un caso, o solo se puede hallar una? De ser así, ¿en qué forma? ¿Y si no es posible, por qué no?

Problema 8.14. Un caso del problema de la mochila que se describe en la Sección 8.4 puede tener diferentes soluciones óptimas. ¿Cómo se descubriría esto? ¿Permite la tabla V recuperar más de una solución en este caso?

Problema 8.15. En la Sección 8.4 asumíamos que se disponía de n objetos, numerados desde 1 hasta n. Supongamos en cambio que se tienen n *tipos* de objetos disponibles, con un suministro adecuado de todos los tipos. Formalmente, esto no es más que sustituir la limitación de que x_i debe de ser 0 o 1 por la limitación menos restrictiva de que x_i debe de ser un entero no negativo. Adaptar el algoritmo de programación dinámica de la Sección 8.4 para resolver este nuevo problema.

Problema 8.16. Adaptar el algoritmo del problema 8.15 para que funcione aun cuando el número de objetos de un tipo dado sea limitado.

Problema 8.17. ¿Funciona el algoritmo de Floyd (véase la Sección 8.5) en un grafo que tenga algunas aristas cuyas longitudes sean negativas, pero que no contenga ningún ciclo negativo? Justifique su respuesta.

Problema 8.18. (Algoritmo de Warshall) Tal como en el algoritmo de Floyd (véase la Sección 8.5) nos interesa hallar caminos dentro de un grafo. En este caso, sin embargo, la longitud de las aristas carece de interés; sólo es importante su existencia. Sea una matriz L tal que $L[i, j]$ = *verdadero* si existe la arista (i, j), y $L[i, j]$ = *falso* en caso contrario. Deseamos encontrar una matriz D tal que $D[i, j]$ = *verdadero* si existe al menos un camino desde i hasta j, y $D[i, j]$ = *falso* en caso contrario. Adaptar el algoritmo de Floyd para este caso ligeramente distinto.

Nota: Estamos buscando el *cierre transitivo reflexivo* del grafo en cuestión.

Problema 8.19. Buscar un algoritmo significativamente mejor para el problema anterior en el caso en que la matriz L es simétrica, esto es, cuando $L[i, j] = L[j, i]$.

Problema 8.20. Esperamos (en vano) encontrar un algoritmo voraz para el problema de la multiplicación encadenada de matrices; véase la Sección 8.6. Suponga que tenemos que calcular

$$M = M_1 M_2 \cdots M_n$$

en donde la matriz M_i es $d_{i-1} \times d_i$, $1 \le i \le n$. Para todas y cada una de las técnicas sugeridas a continuación, proporcionar un contraejemplo en el que no funcione la técnica en cuestión.

(a) Multiplicar primero las matrices M_i y M_{i+1} cuya dimensión común d_i sea la más pequeña, y seguir de igual manera.

(b) Multiplicar primero las matrices M_i y M_{i+1} cuya dimensión común d_i sea la mayor, y seguir de igual manera.

(c) Multiplicar primero las matrices M_i y M_{i+1} que minimicen el producto $d_{i-1}d_i d_{i+1}$, y seguir de igual manera.

(d) Multiplicar primero las matrices M_i y M_{i+1} que maximicen el producto $d_{i-1}d_i d_{i+1}$, y seguir de igual manera.

Problema 8.21. Escriba con detalle el algoritmo para calcular los valores de m_{ij} descrito en la Sección 8.6.

Problema 8.22. Adapte el algoritmo del problema anterior para que no sólo calcule m_{ij} sino que además diga la forma en que hay que calcular el producto de matrices para alcanzar el valor óptimo de m_{1n}.

Problema 8.23. ¿Cuál es el error del siguiente razonamiento sencillo? «El algoritmo para calcular los valores de m dado en la Sección 8.6 tiene que rellenar, en esencia, las entradas de poco más de la mitad de una tabla $n \times n$. Su tiempo de ejecución, por tanto, está claramente en $\Theta(n^2)$.»

Problema 8.24. Sea $T(n)$ un número de Catalán; véase la Sección 8.6. Demostrar que

$$T(n) = \frac{1}{n}\binom{2n-2}{n-1}.$$

Problema 8.25. Demostrar que el número de formas de cortar un polígono convexo de n lados en $n-2$ triángulos, empleando líneas diagonales que no se crucen, es $T(n-1)$, el $(n-1)$-ésimo número de Catalán; véase la Sección 8.6. Por ejemplo, un hexágono se puede cortar de 14 maneras diferentes, según se muestra en la figura 8.7.

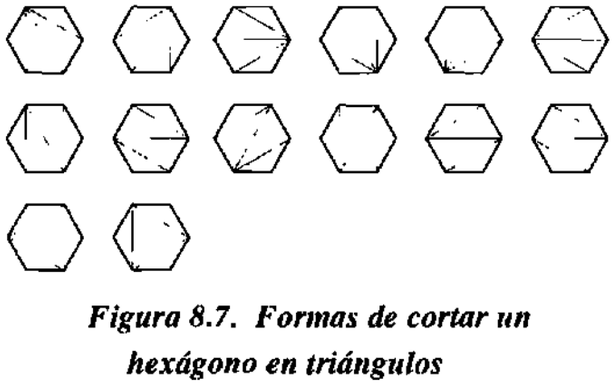


Figura 8.7. Formas de cortar un hexágono en triángulos

Problema 8.26. Mostrar la forma de calcular (i) un coeficiente binomial y (ii) la función *serie*(n, p) de la Sección 8.1.2 empleando una función con memoria.

Problema 8.27. Mostrar la forma de resolver (i) el problema de dar la vuelta y (ii) el problema de la mochila de la Sección 8.4, empleando una función con memoria.

Problema 8.28. Considere el alfabeto $\sum = \{a, b, c\}$. Los elementos de $\sum$ tienen la siguiente tabla de multiplicación, donde las filas muestran el símbolo de la izquierda y las columnas muestran el símbolo de la derecha.

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

De esta manera, $ab = b$, $ba = c$ y así sucesivamente. Observe que la multiplicación definida por esta tabla no es conmutativa ni asociativa.

Buscar un algoritmo eficiente que examine una cadena $x = x_1x_2 \ldots x_n$ de caracteres de $\sum$, y decida si es posible o no poner paréntesis en x de tal manera que el valor de la expresión resultante sea a. Por ejemplo, si $x = bbbba$, el algoritmo debería de responder «si» porque $(b(bb))(ba) = a$. Esta expresión no es única. Por ejemplo, $(b(b(b(ba)))) = a$ también. En términos de n, la longitud de la cadena x, ¿cuánto tiempo requiere este algoritmo?

Problema 8.29. Modifique el algoritmo del ejemplo anterior para que devuelva el número de formas distintas de poner paréntesis en x para obtener a.

Problema 8.30. Sean u y v dos cadenas de caracteres. Deseamos transformar u en v con el menor número posible de operaciones de los tres tipos siguientes: borrar un carácter, añadir un carácter o modificar un carácter. Por ejemplo, podemos transformar *abbac* en *abcbc* en tres etapas:

abbac	→ *abac*	(borrar *b*)
	→ *ababc*	(añadir *b*)
	→ *abcbc*	(transformar *a* en *c*)

Demostrar que esta transformación no es óptima.

Escribir un algoritmo de programación dinámica que busque el número mínimo de operaciones necesarias para transformar u en v y que nos diga cuáles son esas operaciones. En función de las longitudes de u y v, ¿cuánto tiempo requiere este algoritmo?

Problema 8.31. Se dispone de n objetos que es necesario ordenar empleando las relaciones «<» y «=». Por ejemplo, con tres objetos se tienen 13 ordenaciones posibles

$$a = b = c \quad a = b < c \quad a < b = c \quad a < b < c \quad a < c < b$$
$$a = c < b \quad b < a = c \quad b < a < c \quad b < c < a \quad b = c < a$$
$$c < a = b \quad c < a < b \quad c < b < a$$

Dar un algoritmo de programación dinámica que pueda calcular, como función de n, el número de ordenaciones posibles. El algoritmo debe necesitar un tiempo que esté en $O(n^2)$, y un espacio que esté en $O(n)$.

Problema 8.32. A lo largo de un río hay n aldeas. En cada aldea se puede alquilar una canoa que se puede devolver en cualquier otra aldea que esté a favor de la corriente (es casi imposible remar en contra de la corriente). Para todo posible punto de partida i, y para todo posible punto de llegada j, se conoce el coste de un alquiler desde i hasta j. Sin embargo, puede ocurrir que el coste del alquiler desde i hasta j sea mayor que el coste total de una serie de alquileres más breves.

En tal caso, se puede devolver la primera canoa en alguna aldea k entre i y j, y seguir camino en una segunda canoa. No hay cargos adicionales por cambiar de canoa de esta manera.

Dar un algoritmo eficiente para determinar el coste mínimo de un viaje en canoa desde todos los posibles puntos de partida i hasta todos los posibles puntos de llegada j. En términos de n, ¿cuánto tiempo requiere su algoritmo?

Problema 8.33. Cuando se describían los árboles binarios de búsqueda en la Sección 5.5, mencionamos que era buena idea mantenerlos equilibrados. Esto es cierto siempre y cuando la probabilidad de acceder a todos los nodos sea la misma. Sin embargo si se accede a algunos nodos con más frecuencia que a otros, un árbol desequilibrado puede ofrecer un mejor rendimiento medio. Por ejemplo, el árbol de la figura 8.8 es mejor que el de la figura 5.9 si nos interesa minimizar el número medio de comparaciones con el árbol, y si se accede a los nodos con las siguientes probabilidades.

Nodo	6	12	18	20	27	34	35
Prob.	0,2	0,25	0,05	0,1	0,05	0,3	0,05

Más generalmente, supongamos que tenemos un conjunto ordenado $c_1 < c_2 < \ldots c_n$ de n claves distintas. La probabilidad de que una consulta se refiera a la clave c_i es p_i, $1 \le i \le n$. Suponga por sencillez que todas las consultas se refieren a claves que están en el árbol de búsqueda, así que $\sum_{i=1}^{n} p_i = 1$. Recuerde que la profundidad de la raíz de un árbol es 0, la profundidad de sus hijos es 1, y así sucesivamente. Si se almacena la clave c_i en un nodo de profundidad d_i, entonces se necesitan d_i+1 comparaciones para hallarla. Por tanto para un árbol dado, el número medio de comparaciones es:

$$C = \sum_{i=1}^{n} p_i(d_i + 1)$$

Por ejemplo, el número medio de comparaciones necesarias con el árbol de la Figura 8.8 es

$$0{,}3 + (0{,}25 + 0{,}05) \times 2 + (0{,}2 + 0{,}1) \times 3 + (0{,}05 + 0{,}05) \times 4 = 2{,}2.$$

(a) Calcular el número medio de comparaciones con el árbol de la figura 5.9, y verificar que el árbol de la figura 8.8 es mejor.

(*b*) El árbol de la figura 8.8 se obtuvo a partir de las probabilidades dadas empleando un algoritmo sencillo. ¿Puede adivinar cuál es?

(*c*) Buscar otro árbol de búsqueda para el mismo conjunto de claves, que sea todavía más eficiente en el caso promedio que el de la figura 8.8. ¿Qué conclusión acerca del diseño de algoritmos pone esto de manifiesto?

Problema 8.34. Continuando con el problema 8.33, diseñar un algoritmo de programación dinámica para hallar un árbol binario de búsqueda óptimo para un conjunto de claves con probabilidades de acceso dadas. ¿Cuánto tiempo requiere su algoritmo en función del número de claves? Aplique su algoritmo al caso dado en el problema 8.33.

Sugerencia: En todo árbol de búsqueda en el que los nodos que contienen las claves $c_i, c_{i+1}, \ldots, c_j$ forman un subárbol, sea C_{ij} el mínimo número medio de accesos que se hacen a esos nodos. En particular, $C_{1,n}$ es el número medio de accesos causados por una consulta a un árbol binario de búsqueda óptimo y $C_{ii} = p_i$ para todo i, $1 \le i \le n$. Invocar el principio de optimalidad para afirmar que

$$C_{ij} = \min_{i \le k \le j} \left(C_{i,k-1} + C_{k+1,j} \right) + \sum_{k=i}^{j} p_k$$

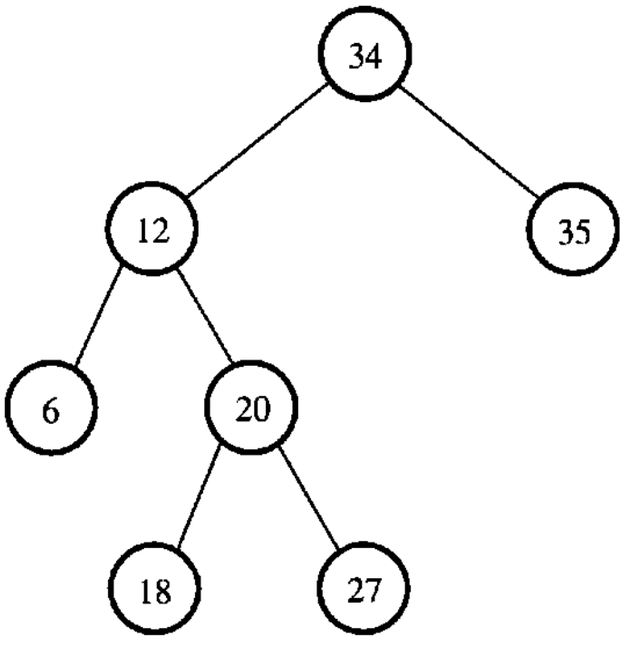


Figura 8.8. Otro árbol de busqueda

cuando $i < j$. Dar un algoritmo de programación dinámica para calcular C_{ij} para todos los $0 \le i \le j \le n$, y un algoritmo para hallar el árbol binario de búsqueda óptimo a partir de los C_{ij}.

Problema 8.35. Resolver de nuevo el problema 8.34. Esta vez, el algoritmo para calcular un árbol binario de búsqueda óptimo para un conjunto de n claves debe funcionar en un tiempo que esté en $O(n^2)$.

Sugerencia: Demuestre primero que $r_{i,j-1} \le r_{ij} \le r_{i+1,j}$ para todos los $1 \le i \le j \le n$, en donde r_{ij} es la raíz de un subárbol de búsqueda óptimo que contiene a $c_i, c_{i+1}, \ldots, c_j$ para todos los $1 \le i \le j \le n$ (los empates se deshacen arbitrariamente) y $r_{i,i-1} = i, 1 \le i \le n$.

Problema 8.36. Como función de n, ¿cuántos árboles binarios de búsqueda existen para n claves diferentes?

Problema 8.37. Recuerde que la función de Ackermann $A(m,n)$ definida en el problema 5.38 crece con suma rapidez. Dar un algoritmo de programación dinámica para calcularla. El algoritmo debe constar simplemente de dos bucles anidados, y no se admite la recursión. Además, se tiene la limitación de utilizar un espacio que esté en $O(m)$ para calcular $A(m, n)$. Sin embargo, se puede suponer que una palabra de almacenamiento puede contener un entero arbitrariamente grande.

Sugerencia: Utilice dos vectores *valor*[0..*m*] e *índice*[0..*m*] y asegúrese de que sea *valor*[*i*] = $A(i,$ *índice*[*i*]) al final de cada pasada por el bucle interno.

8.10 REFERENCIAS Y TEXTOS MÁS AVANZADOS

Hay varios libros que abarcan la programación dinámica. Mencionaremos únicamente a Bellman (1957), Bellman y Dreyfus (1962), Nemhauser (1966) y Laurière (1979).

El algoritmo de la Sección 8.2 para devolver cambio se discute en Wright (1975) y en Chang y Korsh (1976). Para más ejemplos de resolución del problema de la mochila empleando programación dinámica, véase Hu (1981).

El algoritmo de la Sección 8.5 para calcular todos los caminos más cortos se debe a Floyd (1962). Se conoce un algoritmo teóricamente más eficiente: Fredman (1976) muestra la forma de resolver el problema en un tiempo que está en $O(n^3 \sqrt[3]{\log \log n / \log n})$. La solución del problema 8.18 se consigue mediante el algoritmo de Warshall (1962). Tanto el algoritmo de Warshall como el de Floyd son en esencia iguales al anterior de Kleene (1956) para determinar la expresión regular que corresponde a un autómata finito; véase Hopcroft y Ullman (1979). Todos estos algoritmos, con la excepción del de Fredman, se unifican en Tarjan (1981).

El algoritmo de la Sección 8.6 para la multiplicación encadenada de matrices se describe en Godbole (1973); un algoritmo más eficiente, que puede resolver el problema en un tiempo que está en $O(n \log n)$ se encuentra en Hu y Shing (1982, 1984). Los números de Catalán se describen

en muchos lugares, incluyendo Sloane (1973) y Purdom y Brown (1985). Las funciones con memoria se introducen en Michie (1968); para más detalles véase Marsh (1970).

El problema 8.25 se discute en Sloane (1973). Una solución del problema 8.30 se da en Wagner y Fischer (1974). El problema 8.31 se les ocurrió a los autores mientras corregían un examen que contenía una pregunta similar al problema 3.21: ¡sentíamos curiosidad por saber qué proporción de todas las posibles respuestas estaba representada por las 69 respuestas distintas sugeridas por los alumnos!

El problema 8.34 acerca de la construcción de árboles binarios de búsqueda óptimos proviene de Gilbert y Moore (1959), lugar en que se amplía a la posibilidad de que la clave solicitada no se encuentre en el árbol. La mejora considerada en el problema 8.35 proviene de Knuth (1971, 1973), pero una solución más sencilla y más general se da en Yao (1980), que también da una condición suficiente para que ciertos algoritmos de programación dinámica se ejecutan en un tiempo cúbico, sean transformables de forma automática en algoritmos de tiempo cuadrático. El árbol de búsqueda óptimo para las 31 palabras más comunes del inglés se compara en Knuth (1973) con el árbol que se obtiene utilizando el algoritmo voraz evidente que se sugiere en el problema 8.33 (b).

Entre los algoritmos importantes de programación dinámica que no hemos mencionado se encuentran el de Kasimi (1965) y Younger (1967) que requiere un tiempo cúbico para efectuar el análisis sistemático de cualquier lenguaje libre de contexto (véase Hopcroft y Ullman 1979), y el de Held y Karp (1962) que resuelve el problema del viajante (véanse las Secciones 12.5.2 y 13.1.2) en un tiempo que está en $O(n^2 2^n)$, mucho mejor que el tiempo en $\Omega(n!)$ que requiere el algoritmo menos sofisticado.

Capítulo 9

Exploración de grafos

Existe una gran cantidad de problemas que se pueden formular en términos de grafos. Por ejemplo, hemos visto el problema del camino más corto y el problema del árbol de recubrimiento mínimo. Para resolver estos problemas, es frecuente que sea necesario examinar todos los nodos o todas las aristas de un grafo. En algunas ocasiones, la estructura del problema es tal que sólo necesitamos visitar algunos de los nodos, o bien algunas de las aristas. Hasta el momento, los algoritmos que hemos visto imponían implícitamente un orden en estos recorridos: se trataba de visitar el nodo más próximo, o la arista más corta, y así sucesivamente. En este capítulo presentamos algunas técnicas generales que se pueden utilizar cuando no se requiere ningún orden concreto en nuestro recorrido.

9.1 GRAFOS Y JUEGOS: INTRODUCCIÓN

Consideremos el juego siguiente. Se trata de una de las muchas variantes de Nim, que también se conoce con el nombre de juego de Marienbad. Inicialmente, hay un montón de cerillas en la mesa, entre los dos jugadores. El primer jugador puede quitar tantas cerillas como desee, salvo que tiene que tomar una como mínimo, y debe dejar una como mínimo. En el montón inicial tiene que haber un mínimo de dos cerillas. En lo sucesivo, el jugador al que corresponda el turno tiene que tomar como mínimo una cerilla, y como máximo el doble del número de cerillas que acaba de tomar su contrincante. Gana el jugador que se quede con la última cerilla. No hay empates.

Supongamos que en una cierta fase del juego se encuentra usted delante de un montón de cinco cerillas. Su oponente acaba de tomar dos cerillas y le toca jugar a usted. Puede tomar una, dos, tres o cuatro cerillas, pero no puede tomar las cinco porque las reglas prohíben tomar más del doble de las que acaba de tomar el contrario. ¿Qué debe hacer?

La mayoría de las personas que juegan a estos juegos empiezan a revisar las posibilidades en sus mentes: «Si cojo cuatro cerillas, le dejo una sola al contrario, así

que la toma él y gana; si cojo tres, dejo dos para mi oponente, y vuelve a tomarlas y gana; si tomo dos, pasa lo mismo; pero si me quedo con una sola, entonces tendrá delante cuatro cerillas y sólo podrá tomar una o dos. En este caso, no gana de inmediato, así que ciertamente es mi mejor movimiento». Sin más que examinar un movimiento por anticipado, en esta situación sencilla el jugador puede determinar lo que debe hacer a continuación. En un ejemplo más complicado, podrían ser posibles varias jugadas. Para seleccionar la mejor, el jugador quizá tenga que considerar no sólo la situación posterior a su propia jugada, sino más adelante para ver cómo puede contraatacar su oponente en cada una de las jugadas posibles. Y entonces quizá tenga que pensar acerca de su propia jugada después de cada posible contraataque, y así sucesivamente.

Para formalizar este proceso de pensamiento anticipatorio, representamos el juego mediante un grafo dirigido. Cada nodo del grafo corresponde a una situación del juego, y cada arista corresponde a una jugada que nos lleva de una situación a otra. (En algunos contextos, por ejemplo en los informes de juegos de ajedrez, cada jugada consta de una acción por parte de un jugador junto con la respuesta por parte de su oponente. En tales contextos, el término *media jugada* se emplea para denotar la acción de un solo jugador. En este libro, nos limitaremos a la terminología sencilla y llamaremos *jugada* a la acción de cualquiera de los jugadores.) Una situación del juego no sólo se especifica por el número de cerillas que queden en la mesa. También es necesario conocer el límite superior del número de cerillas que se pueden quitar en la jugada siguiente. Sin embargo, no es necesario conocer a quién le toca jugar, puesto que las reglas son las mismas para ambos jugadores (a diferencia de juegos tales como cacos y polis, en que los jugadores tienen objetivos y fuerzas diferentes). Los nodos del grafo que corresponde a este juego son por tanto parejas de la forma $\langle i, j \rangle$. En general, $\langle i, j \rangle$ con $1 \leq j \leq i$ indica que en la mesa quedan i cerillas, y que en la jugada siguiente se puede tomar cualquier número de cerillas entre 1 y j. Las aristas que salen de esta situación, esto es, las jugadas que se pueden hacer, van a los j nodos $\langle i-k, \text{mín}(2k, i-k) \rangle$, $1 \leq k \leq j$. El nodo que corresponde a la posición inicial de un juego con n cerillas es $\langle n, n-1 \rangle$, $n \geq 2$. La posición $\langle 0, 0 \rangle$ da lugar a perder el juego: si un jugador se encuentra en esta situación cuando le toca mover, su oponente acaba de tomar la última cerilla, y ha ganado.

La figura 9.1 muestra parte del grafo correspondiente al juego. De hecho, se trata de la parte del grafo que necesita el jugador del ejemplo anterior, que se enfrenta a un montón de cinco cerillas de las cuales puede tomar cuatro: es la situación $\langle 5, 4 \rangle$. No aparece ninguna situación de la forma $\langle i, 0 \rangle$ salvo por la situación de derrota, $\langle 0, 0 \rangle$. Estas situaciones no se pueden alcanzar durante una partida, así que no tienen interés. De forma análoga, los nodos $\langle i, j \rangle$ con j impar y $j < i-1$ no se pueden alcanzar desde ninguna situación inicial, así que también se omiten. Tal como explicaremos dentro de poco, los nodos cuadrados representan situaciones de derrota y los nodos redondos denotan situaciones de victoria. Las aristas gruesas corresponden a jugadas victoriosas: desde una situación de victoria, ganaremos al seleccionar una de las aristas gruesas. En las situaciones de derrota no hay aristas

gruesas que salgan de ellas, lo cual corresponde al hecho de que en estas situaciones no hay ninguna jugada que nos dé la victoria. Observamos que el jugador que tiene que mover primero en un juego con dos, tres o cinco cerillas no tiene ninguna estrategia que le lleve a la victoria, mientras que en el juego de cuatro cerillas sí que dispone de una estrategia adecuada.

Para decidir cuáles son las situaciones de victoria y cuáles son las situaciones de derrota, partimos de la situación de derrota ⟨0, 0⟩ y retrocedemos. Este nodo no tiene sucesor, y el jugador que se encuentre en esta situación perderá la partida. En cualquiera de los nodos ⟨1, 1⟩, ⟨2, 2⟩ o ⟨3, 3⟩, el jugador puede hacer una jugada que pondrá a su oponente en una situación de derrota. Estos tres nodos, por tanto, son los nodos de victoria. Partiendo de ⟨2, 1⟩, la única jugada posible es pasar a ⟨1, 1⟩. En la situación ⟨2, 1⟩, por tanto, el jugador está obligado a poner a su oponente en una situación de victoria, así que ⟨2, 1⟩ es una situación de derrota. A la posición de derrota ⟨3, 2⟩ se le puede aplicar un argumento similar. Hay dos jugadas posibles, pero ambas dejan al oponente en una situación de victoria, así que ⟨3, 2⟩ es una situación de derrota. Partiendo de ⟨4, 2⟩ o de ⟨4, 3⟩ existe una jugada que pone al oponente en una situación de derrota, a saber, ⟨3, 2⟩; por tanto, estos dos nodos son situaciones de victoria. Por último, las cuatro jugadas posibles desde ⟨5, 4⟩ dejan todas ellas al contrario en situación de victoria, así que ⟨5, 4⟩ es una situación de derrota.

En un grafo más grande, este proceso de etiquetado de situaciones de victoria y de derrota se puede prolongar hacia atrás cuanto se quiera. Las reglas que hemos estado aplicando se pueden resumir como sigue: una situación será una situación

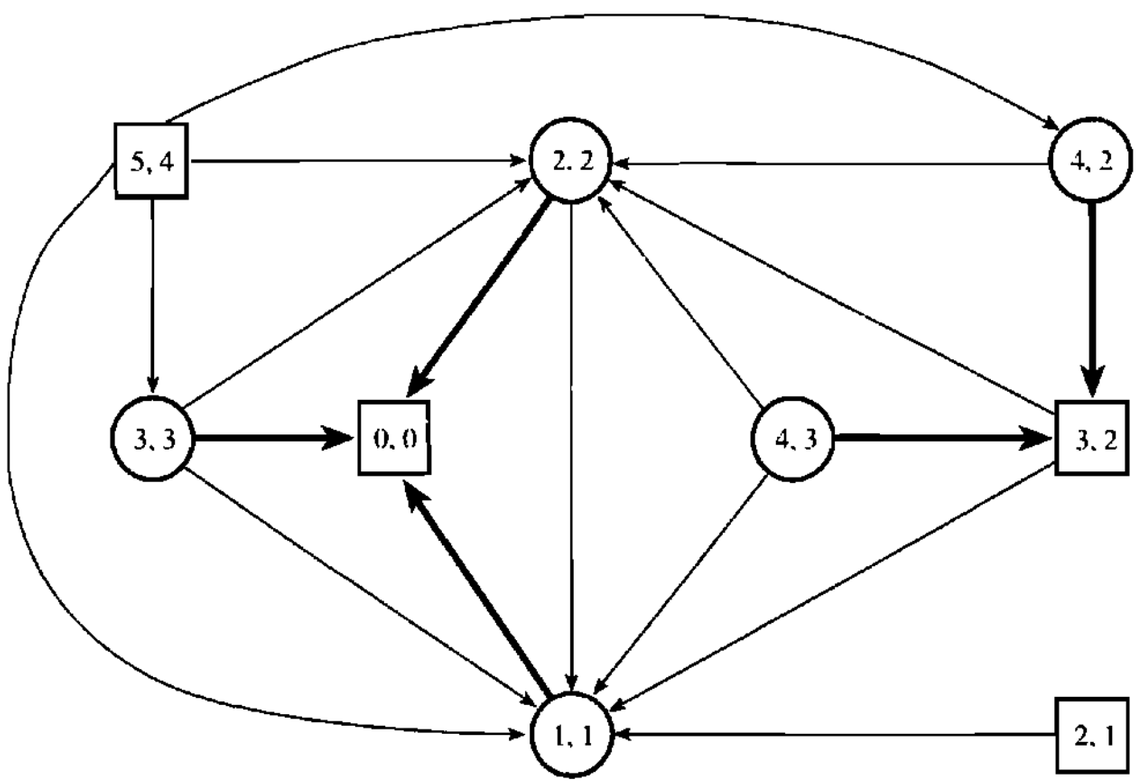


Figura 9.1. Parte del grafo de un juego

de victoria si al menos *uno* de sus sucesores es una situación de derrota, porque entonces el jugador puede jugar para poner a su oponente en una situación de derrota; una posición es de derrota si *todos* sus sucesores son situaciones de victoria, porque el jugador no puede evitar dejar a su oponente en una situación de victoria. El algoritmo siguiente, por tanto, determina si una situación es de victoria o de derrota:

```
función ganarec(i, j)
    {Devuelve verdadero si y sólo si la situación ⟨i, j⟩ es de victoria,
    suponemos que 0 ≤ j ≤ i}
    para k ← 1 hasta j hacer
        si no ganarec(i − k, mín(2k,i −k))
        entonces devolver verdadero
    devolver falso
```

Este algoritmo adolece de los mismos defectos que el algoritmo *Fibrec* de la Sección 2.7.5: calcula el mismo valor una y otra vez. Por ejemplo, *ganarec*(5, 4) proporciona *falso*, después de haber llamado sucesivamente a *ganarec*(4, 2), *ganarec*(3, 3), *ganarec*(2, 2) y *ganarec*(1, 1), pero *ganarec*(3, 3) también llama a *ganarec*(2, 2) y a *ganarec*(1, 1).

Para eliminar esta ineficiencia hay dos enfoques evidentes. El primero, que consiste en aplicar programación dinámica, nos obliga a crear una matriz booleana G tal que $G[i, j]$ = *verdadero* si y sólo si $<i, j>$ es una situación de victoria. Como es habitual con la programación dinámica, procederemos de forma ascendente, calculando $G[r, s]$ para $1 \leq s \leq r < i$, así como los valores de $G[i, s]$ para $1 \leq s < j$, antes de calcular $G[i, j]$:

```
procedimiento ganadin(n)
    { Para cada 1 ≤ j ≤ i ≤ n da a G[i, j] el valor verdadero
    si y sólo si la situación ⟨i, j⟩ es de victoria}
    G[0, 0] ← falso
    para i ← 1 hasta n hacer
        para j ← 1 hasta i hacer
            k ← 1
            mientras k < j y G[i − k, mín(2k, i − k)] hacer
            k ← k + 1
        G[i, j] ← no G[i − k, mín(2k, i − k)]
```

En este contexto la programación dinámica nos lleva a calcular de forma innecesaria algunas entradas de la matriz G que no se llegan a utilizar nunca. Por ejemplo, sabemos que ⟨15, 14⟩ es una situación de victoria en cuanto descubrimos que su segundo sucesor ⟨13, 4⟩ es una situación de derrota. Ya no nos interesa saber si el siguiente sucesor, ⟨12, 6⟩ es una situación de victoria o de derrota. De hecho, sólo son útiles 28 nodos cuando calculamos $G[15, 14]$, a pesar de que el algoritmo de progra-

mación dinámica determina 121 de estos nodos. Se puede ahorrar aproximadamente la mitad de este trabajo si no se calcula $G[i, j]$ cuando j es impar y $j < i - 1$, puesto que estos nodos nunca son interesantes, pero no hay ninguna razón «ascendente» para no calcular $G[12, 6]$. Como de costumbre, las cosas empeoran a medida que va creciendo el caso: para resolver el juego con 248 cerillas basta con explorar 1.000 nodos, y sin embargo *ganadin* examina más de treinta veces esta cantidad.

El algoritmo recursivo dado anteriormente es ineficiente porque recalcula varias veces el mismo valor. Como consecuencia de su naturaleza descendente, sin embargo, nunca calcula valores innecesarios. Una solución que combina las ventajas de ambos algoritmos consiste en utilizar una función con memoria; véase la Sección 8.8. Esto necesita recordar qué nodos hemos visitado ya durante el cálculo recursivo, empleando una matriz booleana *conocido*[0..*n*, 0..*n*], donde n es una cota superior del número de cerillas que se van a utilizar. Las inicializaciones necesarias son las siguientes:

```
G[0, 0] ← falso, conocido[0, 0] ← verdadero
        para i ← 1 hasta n hacer
            para j ← 1 hasta i hacer
                conocido[i, j] ← falso
```

A continuación, para descubrir si $\langle i, j \rangle$ es una situación de victoria o de derrota, se invoca la función siguiente:

```
función nim(i, j)
        {Para todo 1 ≤ j ≤ i ≤ n, proporciona verdadero
        si y sólo si la situación <i, j> es ganadora}
        si conocido[i, j] entonces devolver G[i, j]
        conocido[i, j] ← verdadero
        para k ← 1 hasta j hacer
              si no nim(i − k, min(2k, i − k)) entonces
              G[i, j] ← verdadero
              devolver verdadero
        G[i, j] ← falso
        devolver falso
```

A primera vista, no hay ninguna razón particular para preferir esta aproximación frente a la programación dinámica, porque en todo caso es preciso invertir un tiempo para inicializar toda la matriz *conocido*[0..*n*, 0..*n*]. Sin embargo, la iniciación virtual (descrita en la Sección 5.1) nos permite evitar esto y obtener una ganancia de eficiencia.

El juego que hemos considerado hasta el momento es tan sencillo que se puede resolver sin utilizar el grafo asociado; véase el Problema 9.5. Sin embargo, estos mismos principios se pueden aplicar a muchos otros juegos de estrategia. Como antes, un nodo de un grafo dirigido corresponde a una situación particu-

lar del juego, y una arista corresponde a una jugada o movimiento válido entre dos situaciones. El grafo es infinito si no hay un límite *a priori* del número de situaciones posibles en el juego. Por sencillez, supondremos que el juego se desarrolla entre dos jugadores que se van turnando, que las reglas son las mismas para los dos jugadores (decimos que el juego es *simétrico*) y que la casualidad no interviene en el resultado (el juego es *determinista*). Las ideas que presentamos se pueden adaptar fácilmente a contextos más generales. Además suponemos que ningún caso del juego puede tener una duración infinita y que ninguna situación del juego ofrece un número infinito de jugadas válidas para el jugador a quien corresponda ese turno. En concreto, algunas situaciones del juego, llamadas *situaciones terminales*, no ofrecen jugadas válidas, y por tanto algunos nodos del grafo no poseen sucesores.

Para determinar una estrategia ganadora en un juego de esta clase, asociamos a cada nodo del grafo una etiqueta seleccionada dentro del conjunto *victoria*, *derrota* y *tablas*. La etiqueta alude a la perspectiva de un jugador que vaya a pasar a la situación correspondiente, suponiendo que ninguno de los jugadores cometa un error. Las etiquetas se asignan de manera sistemática de la forma siguiente (en el ejemplo sencillo dado anteriormente, no eran posibles las tablas, así que la etiqueta *tablas* no llegó a utilizarse nunca, y las reglas establecidas no estaban completas):

1. Se etiquetan las posiciones terminales. Las etiquetas asignadas dependen del juego en cuestión. Para la mayoría de los juegos, si nos encontramos en una situación terminal, entonces no podemos hacer ningún movimiento, y hemos perdido. Si embargo, esto no siempre es así. Por ejemplo, si uno no puede mover por bloqueo mutuo en el juego de ajedrez, el resultado son tablas. Y además muchos juegos de Nim vienen por parejas, en una de las cuales gana el jugador que toma la última cerilla, y otra (que se llama versión *misère* del juego) en la que pierde el jugador que se quede con la última cerilla.
2. Una situación no terminal es una situación de victoria si al menos *uno* de los sucesores es una situación de derrota, porque el jugador al que corresponda el turno puede dejar a su oponente en esta situación de derrota.
3. Una situación no terminal es una situación de derrota si *todos* sus sucesores son situaciones de victoria, porque el jugador al que corresponda el turno no puede evitar dejar a su oponente en una de estas posiciones de victoria.
4. Cualquier otra situación no terminal da lugar a tablas. En este caso, entre los sucesores debe contarse al menos una situación de tablas, posiblemente con algunas situaciones adicionales que sean de victoria. El jugador al que corresponda el turno puede evitar dejar a su oponente en una situación de victoria, pero no puede forzarle a una situación de derrota.

Una vez que se han asignado las etiquetas, puede uno leer en el grafo la estrategia para la victoria.

En principio, esta técnica es aplicable incluso a juegos tan complejos como el ajedrez. A primera vista, el grafo asociado al ajedrez parece contener ciclos, puesto que si dos situaciones u y v de las piezas difieren tan sólo en el desplazamiento legal de un caballo, por ejemplo, y el rey no está en jaque, entonces podemos mover igualmente bien desde u hasta v y desde v hasta u. Sin embargo, este problema desaparece tras un examen más detallado. En la variante de Nim que se utilizaba como ejemplo anteriormente, una situación se define no sólo por el número de cerillas que hay en la mesa, sino también por una información invisible que da el número de cerillas que se pueden tomar en la próxima jugada. De manera similar, una situación de ajedrez no queda definida simplemente por la posición de las piezas. También necesitamos saber a quién le corresponde jugar, cuáles son los caballos y reyes que se han movido desde el principio del juego (para saber si es legal enrocar) y si algún peón acaba de avanzar dos casillas (para saber si es posible una captura *al paso*). También hay reglas diseñadas explícitamente para evitar que una partida no acabe nunca. Por ejemplo, se dice que una partida ha quedado en tablas después de que se haya producido un cierto número de jugadas en las que no se haya producido ninguna acción irreversible (el desplazamiento de un peón o una captura). Gracias a estas reglas y otras similares, no hay ciclos en el grafo correspondiente al ajedrez. Sin embargo, esto nos obliga a incluir cosas tales como el número de jugadas habido desde la última acción irreversible en la información que define una situación.

Adaptando las reglas generales que se dan más arriba, podemos etiquetar los nodos como situaciones de victoria para las blancas, situaciones de victoria para las negras, o bien tablas. Una vez construido, este grafo nos permite en principio jugar una partida perfecta de ajedrez, esto es, ganar siempre que sea posible, y perder sólo cuando sea inevitable. Desafortunadamente —o quizá afortunadamente para el juego de ajedrez— el grafo contiene tantos nodos que es totalmente inviable explorarlo en su totalidad, ni siquiera con las computadoras más rápidas existentes en la actualidad. Lo mejor que se puede hacer es explorar el grafo en las proximidades de la situación actual, para ver cómo puede evolucionar la situación, del mismo modo que un novato que razona «si hago esto, él responderá de esta manera, y entonces puedo hacer esto», y así sucesivamente. Sin embargo, ni siquiera esta técnica carece de sutilezas. ¿Debemos examinar *todas* las posibilidades ofrecidas por la situación actual, y para cada una de ellas, *todas* las posibilidades de respuesta? O más bien, ¿debemos seleccionar una línea de ataque prometedora y seguirla durante varias jugadas para ver adónde nos lleva? Las distintas estrategias de búsqueda pueden llevarnos a resultados muy diferentes, como veremos en breve.

Si no tenemos esperanzas de poder explorar todo el grafo para el juego de ajedrez, entonces tampoco podemos tener esperanzas de construirlo y almacenarlo. Lo más que podemos esperar es construir ciertas partes del grafo sobre la marcha, guardándolas si parecen interesantes y descartándolas en caso contrario. Por tanto, a lo largo de este capítulo utilizaremos la palabra «grafo» de dos maneras diferentes.

Por una parte, un grafo puede ser una estructura de datos en la memoria de una computadora. En este caso, los nodos están representados por un cierto número de *bytes*, y las aristas se representan mediante punteros. Las operaciones que hay que efectuar son bastante concretas: «marcar un nodo» significa modificar un *bit* de memoria, «hallar un nodo vecino» significa seguir un puntero, y así sucesivamente.

En otras ocasiones, el grafo solamente existe de forma implícita, tal como sucede cuando exploramos el grafo abstracto correspondiente al juego de ajedrez. Este grafo nunca llega a existir en la memoria de la máquina. La mayor parte del tiempo, lo único que se tiene es una representación de la situación actual (esto es, del nodo que estamos visitando, porque como hemos visto los nodos corresponden a situaciones de las piezas más una cierta información adicional), y posiblemente las representaciones de un pequeño número de situaciones diferentes. Por supuesto, también conocemos las reglas del juego en cuestión. En este caso, «marcar un nodo» significa tomar las medidas adecuadas que nos permiten reconocer una situación que ya hayamos visto, o evitar llegar dos veces a la misma situación. «Encontrar un nodo vecino» significa cambiar la situación actual realizando un único movimiento válido, porque si es posible pasar de una situación a otra efectuando un único movimiento, entonces existe una arista en el grafo implícito que une los dos nodos correspondientes.

Se pueden aplicar unas consideraciones exactamente iguales cuando se explora cualquier grafo de gran tamaño, tal como se verá especialmente en la Sección 9.6. Sin embargo, tanto si el grafo es una estructura de datos, como si es una abstracción que nunca podemos manipular en su conjunto, las técnicas que se utilizan para recorrerlo son esencialmente las mismas. En este capítulo, por tanto, no distinguiremos entre ambos casos.

9.2 RECORRIDOS DE ÁRBOLES

No vamos a invertir demasiado tiempo en describir detalladamente la forma en que se recorren los árboles. Nos limitamos a recordar al lector que en el caso de los árboles binarios se suelen emplear tres técnicas. Si en cada nodo del árbol visitamos primero el nodo, después el subárbol izquierdo y por último el subárbol derecho, entonces estamos recorriendo el árbol en *preorden*; si visitamos el subárbol izquierdo, luego el nodo y finalmente el subárbol derecho, estamos recorriendo el árbol en *orden infijo*; y si visitamos primero el subárbol izquierdo, luego el subárbol derecho y por último el nodo, entonces estamos recorriendo el árbol en *postorden*. El preorden y el postorden se generalizan de forma evidente para árboles no binarios.

Estas tres técnicas exploran el árbol de izquierda a derecha. Hay tres técnicas correspondientes que exploran el árbol de derecha a izquierda. La implementación de cualquiera de estas técnicas utilizando recursión es inmediata.

Lema 9.2.1. *Para cada una de las seis técnicas mencionadas, el tiempo* $T(n)$ *que se necesita para explorar un árbol binario que contenga n nodos se encuentra en* $\Theta(n)$

Demostración. Supongamos que visitar un nodo requiere un tiempo que está en $O(1)$, esto es, que el tiempo requerido está acotado superiormente por alguna constante c. Sin pérdida de generalidad, podemos suponer que $c \geq T(0)$. Supongamos además que tenemos que explorar un árbol que contiene n nodos, con $n > 0$; uno de estos nodos es la raíz, así que si g de ellos se encuentran en el subárbol izquierdo, entonces hay $n - g - 1$ en el subárbol derecho. Entonces

$$T(n) \leq \max_{0 \leq g \leq n-1} \big(T(g) + T(n-g-1) + c\big), \qquad n > 0.$$

Esto es cierto sea cual fuere el orden en que se exploran los subárboles izquierdo y derecho y la raíz. Demostramos por inducción constructiva que $T(n) \leq an + b$, donde a y b son constantes adecuadas, desconocidas por el momento. Si tomamos $b \geq c$, la hipótesis es verdadera para $n = 0$ porque $c \geq T(0)$. Como paso de inducción, sea $n > 0$ y supongamos que la hipótesis es cierta para todo m, con $0 \leq m \leq n - 1$. Entonces

$$\begin{aligned} T(n) &\leq \max_{0 \leq g \leq n-1} \big(T(g) + T(n-g-1) + c\big) \\ &\leq \max_{0 \leq g \leq n-1} \big(ag + b + a(n-g-1) + b + c\big) \\ &\leq an + 3b - a \end{aligned}$$

Entonces, supuesto que tomemos $a \geq 2b$, tenemos $T(n) \leq an + b$, así que la hipótesis es cierta también para $m = n$. Esto demuestra que $T(n) \leq an + b$ para todo $n \geq 0$, y por tanto que $T(n)$ está en $O(n)$.

Por otra parte, está claro que $T(n)$ está en $\Omega(n)$ puesto que se visitan todos y cada uno de los n nodos. Por tanto $T(n)$ está en $\Theta(n)$.

9.2.1 Precondicionamiento

Si tenemos que resolver varios casos parecidos del mismo problema, puede merecer la pena invertir una cierta cantidad de tiempo en calcular resultados auxiliares que puedan ser utilizados en el futuro para acelerar la resolución de cada caso. Esto es el *precondicionamiento*. Informalmente, sea a el tiempo que se necesita para resolver un caso típico cuando *no* se dispone de información auxiliar, sea b el tiempo que se necesita para resolver un caso típico cuando *sí* se dispone de información auxiliar, y sea p el tiempo que se necesita para calcular esta información auxiliar. Para resolver n casos típicos se necesita un tiempo na, sin precondicionamiento y un tiempo $p + nb$ si se emplea precondicionamiento. Siempre que $b < a$, resulta ventajoso utilizar el precondicionamiento cuando $n > p/(a-b)$.

Desde este punto de vista, el algoritmo de programación dinámica para devolver cambio que se daba en la Sección 2.8 se puede ver como un ejemplo de precondicionamiento. Una vez que se han calculado los valores c_m necesarios, podemos dar rápidamente el cambio siempre que sea necesario.

Aun cuando sean pocos los casos que hay que resolver, el cálculo anticipado de información auxiliar puede resultar útil. Supongamos que algunas veces tenemos

que resolver un cierto caso de un conjunto grande de casos posibles. Cuando se necesita una solución, es preciso proporcionarla muy rápidamente, por ejemplo para asegurar una respuesta suficientemente rápida para una aplicación de tiempo real. En este caso, puede no ser práctico calcular y almacenar por anticipado la solución de todos los casos relevantes. Por otra parte, quizá sea posible calcular y almacenar la información auxiliar suficiente para acelerar la resolución de cualquier caso que pudiera aparecer. Una aplicación de precondicionamiento como ésta puede tener importancia práctica aun cuando sólo se resuelva un caso crucial en toda la vida del sistema: podría ser el caso que nos permita, por ejemplo, detener un reactor desbocado.

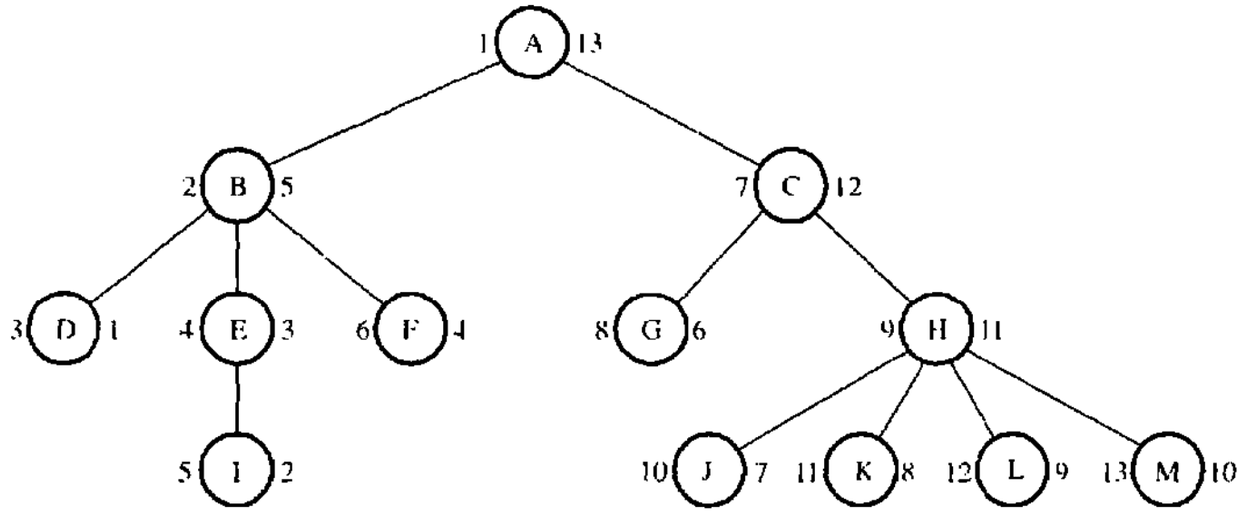


Figura 9.2. Un árbol con raíz, con las numeraciones de preorden y de postorden

Como segundo ejemplo de esta técnica usaremos el problema de determinar los antecesores dentro de un árbol con raíz. Sea T un árbol con raíz, no necesariamente binario. Decimos que un nodo v de T es un *antecesor* del nodo w si v está en el camino que va desde w hasta la raíz de T. En particular, todo nodo es su propio antecesor, y la raíz es un antecesor de todos los nodos. (Aquellos que disfruten con las definiciones recursivas pueden preferir la siguiente: todo nodo es su propio antecesor y, recursivamente, es antecesor de todos los nodos de los cuales sus hijos sean antecesores.) El problema es, dado un par de nodos (v, w) de T, determinar si v es o no un antecesor de w. Si T contiene n nodos, toda solución directa de este caso requiere un tiempo que está en $\Omega(n)$ en el caso peor. Sin embargo, es posible efectuar un precondicionamiento de T en un tiempo que está en $\Theta(n)$ para que en lo sucesivo se pueda resolver cualquier caso particular del problema de la ascendencia en un tiempo constante.

Ilustraremos esta idea usando el árbol de la figura 9.2. Contiene 13 nodos. Para precondicionar el árbol, lo recorremos primero en preorden y después en post-

orden, numerando secuencialmente los nodos a medida que los visitamos. Para un nodo v, sea $prenum[v]$ el número asignado a v cuando se recorre el árbol en preorden, y sea $postnum[v]$ el número que se le asigna durante el recorrido en postorden. En la figura 9.2, estos números aparecen a la izquierda y a la derecha del nodo, respectivamente.

Sean v y w dos nodos del árbol. En preorden, primero numeramos un nodo y después numeramos sus subárboles de izquierda a derecha. Por tanto:

$$prenum[v] \leq prenum[w] \quad \Leftrightarrow \quad \begin{array}{l} v \text{ es un antecesor de } w, \textbf{ o bien} \\ v \text{ está a la izquierda de } w \text{ en el árbol} \end{array}$$

En postorden, primero numeramos los subárboles del nodo de izquierda a derecha, y después numeramos el nodo en sí. Por tanto:

$$postnum[v] \geq postnum[w] \quad \Leftrightarrow \quad \begin{array}{l} v \text{ es un antecesor de } w, \textbf{ o bien} \\ v \text{ está a la derecha de } w \text{ en el árbol} \end{array}$$

De esto se sigue que

$$prenum[v] \leq prenum[w] \text{ y } postnum[v] \geq postnum[w] \Leftrightarrow v \text{ es un antecesor de } w$$

Una vez que los valores de *prenum* y *postnum* se han calculado en un tiempo que está en $\Theta(n)$, la condición requerida se puede comprobar en un tiempo que está en $\Theta(1)$.

9.3 RECORRIDO EN PROFUNDIDAD: GRAFOS NO DIRIGIDOS

Sea $G = \langle N, A \rangle$ un grafo no dirigido formado por todos aquellos nodos que deseamos visitar. Supongamos que de alguna manera es posible marcar un nodo para mostrar que ya ha sido visitado.

Para efectuar un recorrido en *profundidad* del grafo, se selecciona cualquier nodo $v \in N$ como punto de partida. Se marca este nodo para mostrar que ya ha sido visitado. A continuación, si hay un nodo adyacente a v que no haya sido visitado todavía, se toma este nodo como nuevo punto de partida y se invoca recursivamente al procedimiento de recorrido en profundidad. Al volver de la llamada recursiva, si hay otro nodo adyacente a v que no haya sido visitado, se toma este nodo como punto de partida siguiente, se vuelve a llamar recursivamente al procedimiento, y así sucesivamente. Cuando están marcados todos los nodos adyacentes a v, el recorrido que comenzara en v ha finalizado. Si queda algún nodo de G que no haya sido visitado, tomamos cualquiera de ellos como nuevo punto de partida, y volvemos a invocar el procedimiento. Se sigue así hasta que estén marcados todos los nodos de G. Véase a continuación el algoritmo recursivo:

```
procedimiento recorridop(G)
    para cada v ∈ N hacer marca[v] ← no visitado
    para cada v ∈ N hacer
        si marca[v] ≠ visitado entonces rp(v)

procedimiento rp(v)
    {El nodo v no ha sido visitado anteriormente}
    marca[v] ← visitado
    para cada nodo w adyacente a v hacer
        si marca[w] ≠ visitado entonces rp(w)
```

El algoritmo se llama de recorrido en profundidad porque inicia tantas llamadas recursivas como sea posible antes de volver de una llamada. La recursión sólo se detiene cuando la exploración del grafo se ve bloqueada y no puede proseguir. En ese momento, la recursión «retrocede» para que sea posible estudiar posibilidades alternativas en niveles más elevados. Si el grafo corresponde a un juego, se puede pensar intuitivamente que esto es una búsqueda que explora el resultado de una estrategia particular con tantas jugadas anticipadas como sea posible, antes de explorar los alrededores para ver qué tácticas alternativas pudieran estar disponibles.

Considere por ejemplo el grafo de la figura 9.3. Si suponemos que los vecinos de un nodo dado se examinan por orden numérico, y que el nodo 1 es el primer punto de partida, entonces la exploración del grafo en profundidad progresa en la forma siguiente:

```
1. rp(1)                    llamada inicial
2.     rp(2)                llamada recursiva
3.         rp(3)            llamada recursiva
4.             rp(6)        llamada recursiva
5.                 rp(5)    llamada recursiva; no se puede continuar
6.     rp(4)                no se ha visitado un vecino del nodo 1
7.         rp(7)            llamada recursiva
8.             rp(8)        llamada recursiva; no se puede continuar
9. no quedan nodos por visitar
```

¿Cuánto tiempo se necesita para explorar un grafo con n nodos y a aristas? Dado que cada nodo se visita exactamente una vez, hay n llamadas al procedimiento *rp*. Cuando visitamos un nodo, examinamos las marcas de los nodos vecinos. Si se representa el grafo de tal manera que la lista de nodos adyacentes tenga un acceso directo (el tipo *grafolista* de la Sección 5.4), entonces este trabajo es proporcional a a en total. El algoritmo, por tanto, requiere un tiempo que está en $\Theta(n)$ para las llamadas al procedimiento, y un tiempo que está en $\Theta(a)$ para inspeccionar las marcas. Por tanto el tiempo de ejecución está en $\Theta(\max(a, n))$.

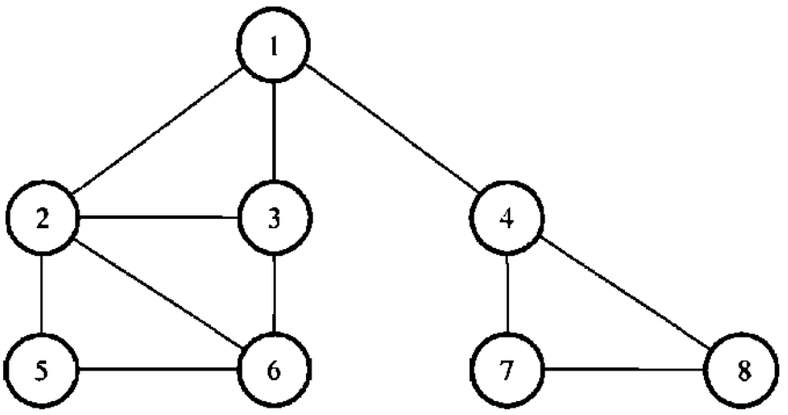


Figura 9.3. Un grafo no dirigido

El recorrido en profundidad de un grafo conexo asocia al grafo un árbol de recubrimiento. Sea T este árbol. Las aristas de T corresponden a las aristas utilizadas para recorrer el grafo; están dirigidas del primer nodo visitado al segundo. Las aristas que no se utilizan en el recorrido del grafo no tienen una arista correspondiente en T. El punto inicial de partida de la exploración pasa a ser la raíz del árbol. Por ejemplo, las aristas utilizadas en el recorrido en profundidad del grafo de la figura 9.3 que se describe más arriba son {1, 2}, {2, 3}, {3, 6}, {6, 5}, {1, 4}, {4, 7} y {7, 8}. Las aristas dirigidas correspondientes, (1, 2), (2,3) y demás, forman un árbol de recubrimiento para este grafo. La raíz del árbol es el nodo 1. El árbol se ilustra en la figura 9.4. Las líneas discontinuas de esta figura corresponden a las aristas de G que no se utilizan en el recorrido en profundidad. Resulta fácil mostrar que una arista de G que no tenga una arista correspondiente en T une necesariamente un nodo v con alguno de sus antecesores en T; véase el problema 9.17.

Si el grafo que se está explorando no es conexo, entonces un recorrido en profundidad le asocia no sólo a un único árbol, sino a todo un bosque de árboles, uno para cada componente conexo del grafo. Un recorrido en profundidad también ofrece una manera de numerar los nodos del grafo que se está visitando. El primer nodo visitado —la raíz del árbol— recibe el número 1, el segundo recibe el número 2, y así sucesivamente. En otras palabras, los nodos del árbol asociado se numeran en preorden. Para implementar esto, hay que añadir las dos sentencias siguientes al principio del procedimiento *rp*:

$nump \leftarrow nump + 1$
$prenum[v] \leftarrow nump$

en donde *nump* es una variable global inicializada a cero. Por ejemplo, el recorrido en profundidad del grafo de la figura 9.3 descrita anteriormente numera los nodos del grafo en la forma siguiente.

nodo	1	2	3	4	5	6	7	8
prenum	1	2	3	6	5	4	7	8

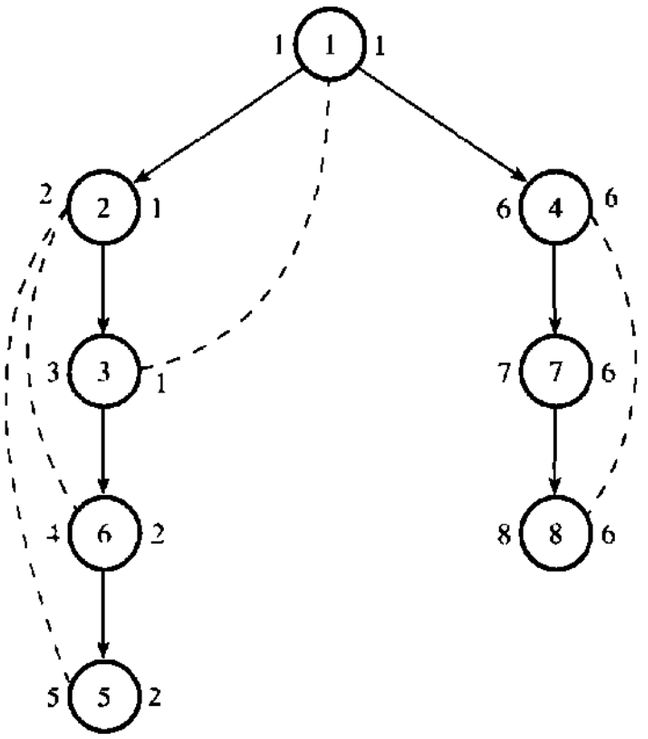


Figura 9.4. Un árbol recorrido en profundidad** de prenum **a la izquierda y** masalto **a la derecha.

Éstos son los números que aparecen a la izquierda de cada nodo en la figura 9.4. Por supuesto, el árbol y la numeración generados por un recorrido en profundidad dentro de un grafo no son únicos, sino que dependen del punto de partida seleccionado y del orden en que se visiten los vecinos.

9.3.1 Puntos de articulación

Un nodo v de un grafo conexo es un *punto de articulación* si el subgrafo que se obtiene borrando v y todas las aristas que incidan en v ya no es conexo. Por ejemplo, el nodo 1 es un punto de articulación del grafo de la figura 9.3; si lo borramos, quedan dos componentes conexas {2, 3, 5, 6} y {4, 7, 8}. Un grafo G se llama *biconexo* (o bien *no articulado*) si es conexo y carece de puntos de articulación. Se denomina *bicoherente* (o bien *carente de istmos*, o *2-arista conexo*) si todo punto de articulación está unido mediante al menos dos aristas con cada componente del subgrafo restante. Estas ideas son importantes en la práctica. Si el grafo G representa, digamos, una red de telecomunicaciones, entonces el hecho de que sea biconexo nos asegura que el resto de la red puede seguir funcionando aunque falle el equipo de uno de los nodos. Si G es bicoherente, podemos estar seguros de que los nodos seguirán pudiendo comunicarse entre sí aunque deje de funcionar una línea de transmisión.

Para ver la forma de hallar los puntos de articulación de un grafo conexo G, examinemos de nuevo la figura 9.4. Recordemos que esta figura muestra *todas* las aristas del grafo G de la figura 9.3: las que se muestran con línea continua forman un árbol T de recubrimiento, mientras que las demás se muestran como líneas discontinuas. Como ya se ha visto, estas otras aristas sólo van desde algún nodo v hasta su antecesor dentro del árbol, y no de una rama a otra. A la izquierda de cada nodo v se encuentra $prenum[v]$, el número asignado mediante un recorrido de T en preorden.

A la derecha del nodo v se encuentra un nuevo número que vamos a llamar *masalto*[v]. Sea w el nodo más alto del árbol que se puede alcanzar desde v siguiendo cero o más líneas continuas, y subiendo entonces como máximo por una línea discontinua. Entonces se define *masalto*[v] como *prenum*[w]. Por ejemplo, desde el nodo 7 podemos bajar por una línea continua hasta el nodo 8, y después subir por una línea discontinua hasta el nodo 4, y éste es el nodo más alto que podemos alcanzar. Dado que *prenum*[4] = 6, también tenemos que *masalto*[7] = 6.

Dado que las líneas discontinuas no cruzan de unas ramas a otras, el nodo más alto alcanzable de esta forma, w, tiene que ser un antecesor de v. (No puede estar por debajo de v dentro del árbol, porque siempre podemos llegar hasta el propio v sin seguir ninguna línea.) Entre los antecesores de v, el más alto dentro del árbol es el que tiene el valor de *prenum* más bajo. Si tenemos estos valores, no es necesario por tanto conocer el nivel exacto de cada nodo: de entre los nodos que podemos alcanzar, nos limitamos a seleccionar aquel que minimice *prenum*.

Consideremos ahora cualquier nodo v de T salvo la raíz. Si v no tiene hijos, no puede ser un punto de articulación de G, porque si lo borramos, los nodos restantes siguen estando conectados mediante las aristas que quedan en T. En caso contrario, sea x un descendiente de v. Supongamos primero que *masalto*[x] < *prenum*[v]. Esto significa que partiendo de x existe una cadena de aristas de G, sin incluir la arista $\{v, x\}$ (porque no está permitido *ascender* por una línea continua), que nos lleva a algún nodo del árbol situado por encima de v. Si borramos v, por tanto, los nodos del subárbol cuya raíz es x no quedarán desconectados del resto del árbol. Esto es lo que sucede con el nodo 3 de la figura, por ejemplo. Aquí, *prenum*[3] = 3, y el único hijo del nodo 3, a saber, el nodo 6, tiene *masalto*[6] = 2 < *prenum*[3]. Por tanto, si borramos el nodo 3, entonces el nodo 6 y sus descendientes seguirán estando conectados a uno de los antecesores del nodo 3.

Si por otra parte *masalto*[x] ≥ *prenum*[v], entonces no hay ninguna cadena de aristas que parta de x (excluyendo de nuevo la arista $\{v, x\}$) y que vuelva a entrar en el árbol por encima de v. En este caso, si se borrase v, los nodos del subárbol cuya raíz es x quedarían desconectados del resto del árbol. El nodo 4 de la figura ilustra este caso. Aquí *prenum*[4] = 6, y el único hijo del nodo 4, a saber, el nodo 7, tiene *masalto*[7] = 6 = *prenum*[4]. Por tanto, si borramos el nodo 4, ningún camino que salga del nodo 7 ó alguno de sus descendientes vuelve a entrar en el árbol por encima del nodo borrado, así que el subárbol cuya raíz está en el nodo 7 se separa del resto de T.

Por tanto, un nodo v que no sea la raíz de T es un punto de articulación de G precisamente si tiene al menos un hijo x con *masalto*[x] ≥ *prenum*[v]. Respecto a la raíz, es evidente que se trata de un punto de articulación de G si y sólo si tiene más de un hijo; en ese caso, dado que no hay aristas que crucen de una rama a otra, al borrar la raíz se desconectan los subárboles restantes de T.

Queda por ver la forma de calcular los valores de *masalto*. Claramente, esto hay que hacerlo desde las hojas hacia arriba. Por ejemplo, desde el nodo 5 podemos quedarnos donde estamos, o subir al nodo 2; éstas son todas las posibilidades. Desde el nodo 6 podemos permanecer donde estamos, subir al nodo 2, o bien bajar primero al nodo 5 y pasar desde allí a donde sea posible llegar, y así sucesivamente. Los

valores de *masalto* se calculan, por tanto, en postorden. En un nodo general v, *masalto*[v] es el mínimo (correspondiente al nodo más alto) de tres clases de valores: *prenum*[v] (nos quedamos donde estábamos), *prenum*[w] para cada nodo w tal que haya una arista $\{v, w\}$ en G sin una arista correspondiente en T (ascendemos por una línea discontinua), y *masalto*[x] para todo hijo x de v (bajamos por una línea continua y vemos hasta dónde se puede llegar por ese camino).

El algoritmo completo para hallar los puntos de articulación de un grafo no dirigido G se resume de la forma siguiente:

1. Se efectúa un recorrido en profundidad en G, empezando por cualquier nodo. Sea T el árbol generado mediante este recorrido, y para cada nodo v de G, sea *prenum*[v] el número que le asigna este recorrido .
2. Se recorre T en postorden. Para cada nodo visitado v, se calcula *masalto*[v] como el mínimo de
   (a) *prenum*[v];
   (b) *prenum*[w] para todo nodo w tal que haya una arista $\{v, w\}$ en G sin una arista correspondiente en T; y
   (c) *masalto*[x] para todo hijo x de v.
3. Se determinan los puntos de articulación de G en la forma siguiente:
   (a) La raíz de T es un punto de articulación precisamente si tiene más de un hijo.
   (b) Cualquier otro nodo v es un punto de articulación si y sólo si tiene un hijo x tal que *masalto*[x] ≥ *prenum*[v].

No es difícil combinar los pasos 1 y 2 del algoritmo anterior, calculando tanto el valor de *prenum* como el de *masalto* durante el recorrido en profundidad de G.

9.4 RECORRIDO EN PROFUNDIDAD: GRAFOS DIRIGIDOS

El algoritmo es esencialmente el mismo que para los grafos no dirigidos; la diferencia reside en la interpretación de la palabra «adyacente». En un grafo dirigido, el nodo w es adyacente al nodo v si existe la arista dirigida (v, w). Si existe (v, w) pero (w, v) no existe, entonces w es adyacente a v, pero v no es adyacente a w. Con este cambio de interpretación, los procedimientos *rp* y *recorridop* de la Sección 9.3 se aplican igualmente bien en el caso de un grafo dirigido.

Sin embargo, el algoritmo se comporta de forma bastante distinta. Consideremos un recorrido en profundidad del grafo dirigido de la figura 9.5. Si los vecinos de un nodo dado se examinan por orden numérico, entonces el algoritmo progresa de la forma siguiente:

```
1. rp(1)        llamada inicial
2.   rp(2)      llamada recursiva
3.     rp(3)    llamada recursiva; no se puede continuar
4.   rp(4)      no se ha visitado un vecino del nodo 1
5.     rp(8)    llamada recursiva
```

6. *rp*(7) llamada recursiva; no se puede continuar
7. *rp*(5) nuevo punto de comienzo
8. *rp*(6) llamada recursiva; no se puede continuar
9. no quedan nodos por visitar

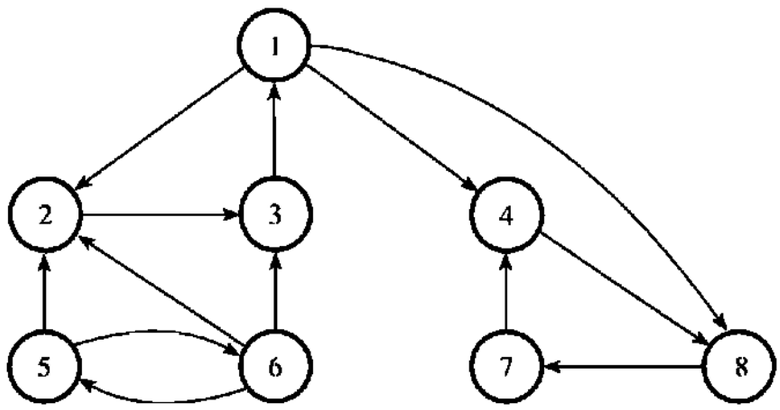


Figura 9.5. Un grafo dirigido

Un argumento idéntico al de la Sección 9.3 muestra que el tiempo que requiere este algoritmo también está en $\Theta(\text{máx}(a, n))$. En este caso, sin embargo, las aristas utilizadas para visitar todos los nodos de un grafo dirigido $G = \langle N, A \rangle$ pueden formar un bosque de varios árboles aunque G sea conexo. Esto es lo que sucede en nuestro ejemplo: las aristas utilizadas, a saber, (1, 2), (2, 3), (1, 4), (4, 8), (8, 7) y (5, 6) forman el bosque mostrado mediante líneas continuas en la figura 9.6.

Sea F el conjunto de aristas que hay en el bosque, de tal manera que $A \setminus F$ es el conjunto de aristas de G que no tienen una arista correspondiente dentro del bosque. En el caso de un grafo no dirigido, veíamos que las aristas del grafo que no tienen una arista correspondiente dentro del bosque unen necesariamente algún nodo con uno de sus antecesores. En el caso de un grafo dirigido, sin embargo, pueden aparecer tres clases de aristas en $A \setminus F$. Se han mostrado mediante líneas discontinuas en la figura 9.6.

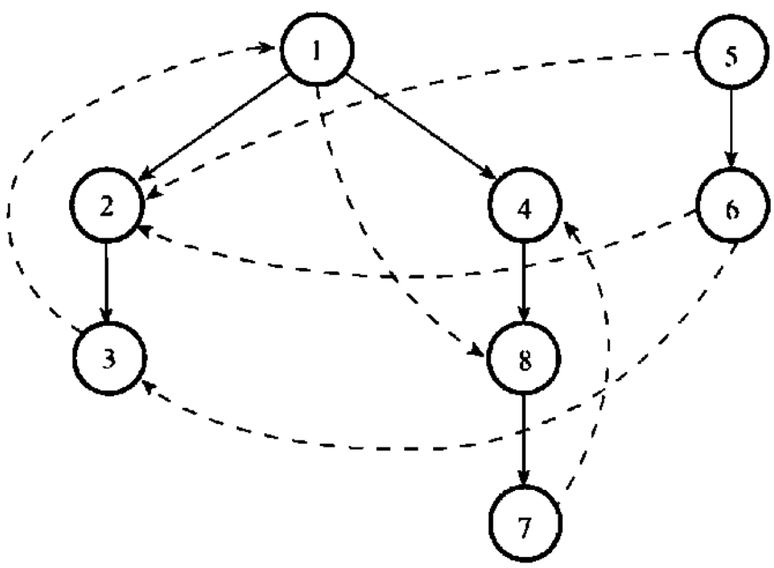


Figura 9.6. Un bosque de recorrido en profundidad

1. Las que, al igual que (3, 1) o (7, 4), van desde un nodo a uno de sus antecesores.
2. Las que, al igual que (1, 8), van de un nodo a uno de sus descendientes.
3. Las que, al igual que (5, 2) o (6, 3), unen un nodo con otro que no es ni antecesor ni descendiente suyo. Las aristas de este tipo están dirigidas, necesariamente, de derecha a izquierda.

9.4.1 Grafos acíclicos: ordenación topológica

Los grafos dirigidos acíclicos se pueden utilizar para representar un cierto número de relaciones interesantes. Esta clase incluye los árboles, pero es menos general que la clase de todos los grafos dirigidos. Por ejemplo, un grafo dirigido acíclico se puede utilizar para representar la estructura de una expresión aritmética que contenga subexpresiones repetidas: de esta manera, la figura 9.7 representa la estructura de la expresión

$$(a + b)(c + d) + (a + b)(c - d)$$

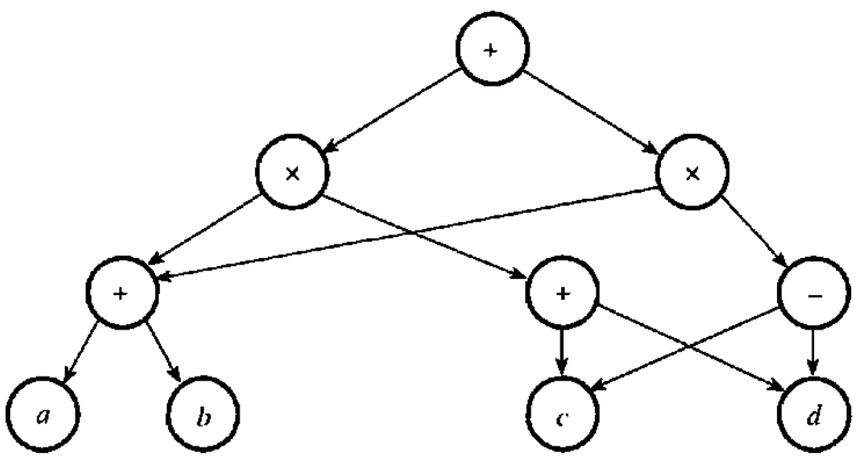


Figura 9.7. Un grafo dirigido acíclico

Estos grafos ofrecen una representación natural para las ordenaciones parciales, como la inclusión de conjuntos. La figura 9.8 ilustra una parte de otra ordenación parcial definida sobre los enteros positivos: aquí hay una arista del nodo i al nodo j si y sólo si i es un divisor propio de j.

Por último, los grafos dirigidos acíclicos suelen utilizarse para especificar la forma en que un proyecto complejo se desarrolla con el tiempo. Los nodos representan distintas fases del proyecto, desde el estado inicial hasta su terminación, y las aristas corresponden a las actividades que es preciso desarrollar para pasar de una fase a otra. La figura 9.9 da un ejemplo de este tipo de diagrama.

Se puede utilizar un recorrido en profundidad para detectar si un grafo dirigido dado es acíclico; véase el problema 9.26. También se puede utilizar para determinar una *ordenación topológica* de los nodos de un grafo dirigido acíclico. En este tipo de ordenación, los nodos del grafo se enumeran de tal manera que si existe una arista (i, j) entonces el nodo i precede al nodo j en la lista. Por ejemplo, para el grafo de la figura 9.8, el orden natural 1, 2, 3, 4, 6, 8, 12, 24 es adecuado, pero por otra

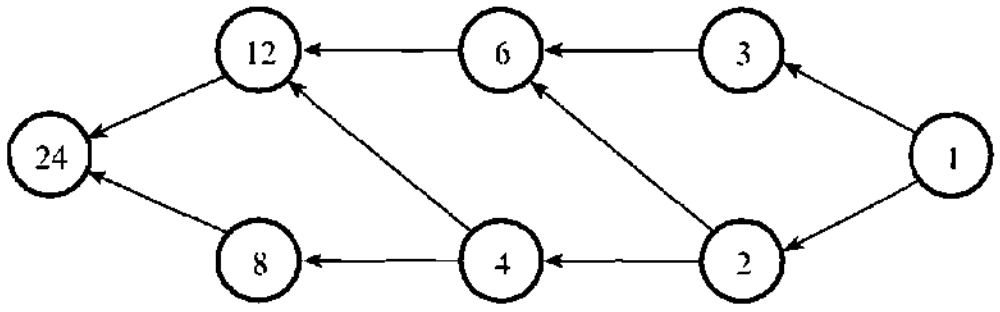


Figura 9.8. Otro grafo dirigido acíclico

parte el orden 1, 3, 2, 6, 4, 12, 8, 24 también es aceptable, e igualmente lo son otros más. Por otra parte, el orden 1, 3, 6, 2, 4, 12, 8, 24 no es válido, porque el grafo contiene una arista (2, 6) y por tanto 2 debe preceder a 6 en la lista.

La adaptación del procedimiento *rp* para hacer que sea una ordenación topológica es sencilla. Añada la línea

escribir v

al final del procedimiento *rp*, se ejecuta el procedimiento para el grafo en cuestión y se invierte entonces el orden de la lista de nodos resultante.

Para ver esto, consideremos lo que sucede cuando llegamos a un nodo v de un grafo dirigido acíclico G, empleando el procedimiento modificado. Algunos de los nodos que deben seguir a v en la ordenación topológica pueden haber sido visitados ya al seguir otra ruta diferente. En este caso, ya se encuentran en la lista tal como deberán estar cuando invirtamos la lista al finalizar la búsqueda. Todo nodo que deba preceder a v en la ordenación topológica se encuentra o bien en el camino que ya estamos explorando, en cuyo caso será marcado como visitado pero no presente en la lista aún, o bien todavía no ha sido visitado. En ambos casos, será añadido a la lista después del nodo v (una vez más, correctamente, porque hay que invertir la lista). Ahora el recorrido en profundidad explora los nodos no visitados que se puedan alcanzar desde v siguiendo las aristas de G. En la ordenación topológica, estos nodos deben ir después de v. Dado que tenemos intención de invertir la lista cuando haya terminado la búsqueda, al añadirlos a la lista durante la exploración que comienza en v, y al añadir v sólo cuando esta exploración haya terminado, obtenemos exactamente lo que necesitábamos.

9.5 RECORRIDO EN ANCHURA

Cuando un recorrido en profundidad llega a un nodo v, intenta a continuación visitar algún vecino de v, después algún vecino del vecino, y así sucesivamente. Cuando un recorrido en anchura llega a algún nodo v, por otra parte, primero visita todos los vecinos de v. Sólo examina nodos más alejados después de haber hecho esto. A diferencia del recorrido en profundidad, el recorrido en anchura no es naturalmente recursivo. Para poner de manifiesto las similitudes y diferencias entre los dos métodos, comenzaremos por dar una formulación no recursiva del al-

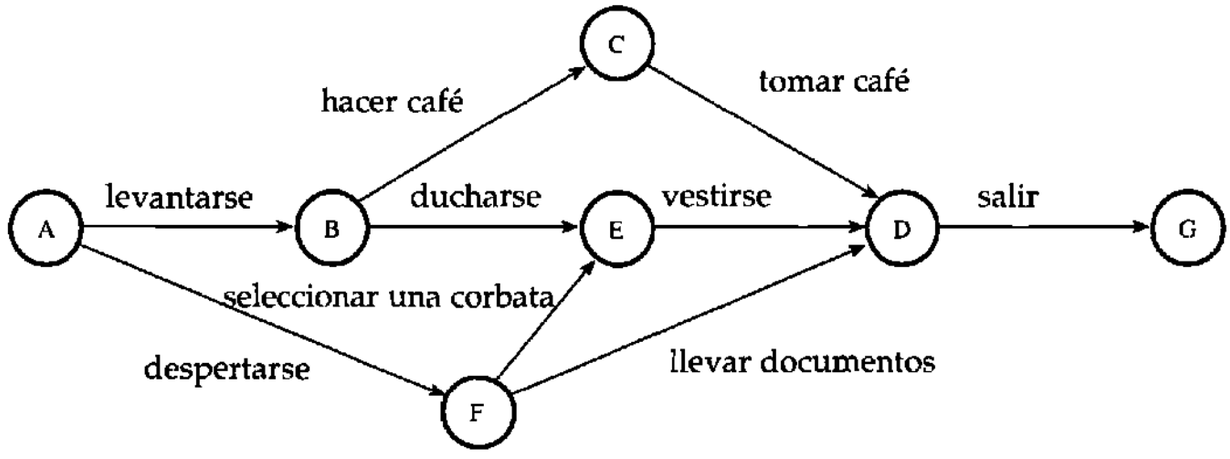


Figura 9.9. Y otro grafo dirigido acíclico

goritmo de recorrido en profundidad. Sea *pila* un tipo de datos que admite dos operaciones, **apilar** y **desapilar**. Se pretende que este tipo represente una lista de elementos que hay que manejar por el orden «último en llegar, primero en salir» (que suele conocerse con el nombre de LIFO, indicando «last in, first out», en inglés). La función *cima* denota el elemento que se encuentra en la parte superior de la pila. Lo que sigue es el algoritmo de recorrido en profundidad ya modificado:

```
procedimiento rp2(v)
    P ← pila vacía
    marca[v] ← visitado
    mientras P no esté vacía hacer
        mientras exista un nodo w adyacente a cima(P)
                tal que marca[w] ≠ visitado
                hacer marca[w] ← visitado
                apilar w en P {w es la nueva cima(P)}
        desapilar P
```

La modificación del algoritmo no ha hecho que cambie su comportamiento. Lo único que hemos hecho es explicitar el apilamiento y desapilamiento de nodos que en la versión anterior era manejado entre bastidores por el mecanismo de pila implícito en cualquier lenguaje recursivo.

Para el algoritmo de recorrido en anchura, en contraste, necesitamos un tipo *cola* que admita las dos operaciones **poner** y **quitar**. Este tipo representa una lista de elementos que hay que manejar por el orden «primero en llegar, primero en salir» (FIFO, que indica «first in, first out»). La función *primero* denota el elemento que ocupa la primera posición de la cola. Véase a continuación el algoritmo de recorrido en anchura:

```
procedimiento ra(v)
    Q ← cola vacía
    marca[v] ← visitado
    poner v en Q
    mientras Q no esté vacía hacer
            u ← primero(Q)
            quitar u de Q
            para cada nodo w adyacente a u hacer
                si marca[w]≠visitado entonces marca[w] ← visitado
                poner w en Q
```

En ambos casos, necesitamos un programa principal que dé comienzo al recorrido:

```
procedimiento recorrido(G)
    para cada v ∈ N hacer marca[v] ← no visitado
    para cada v ∈ N hacer
        si marca[v]≠visitado entonces {bp2 o ra} (v)
```

Por ejemplo, en el grafo de la figura 9.3, si la búsqueda comienza en el nodo 1, y los vecinos de cada nodo se visitan por orden numérico, entonces el recorrido en anchura se desarrolla en la forma siguiente:

	Nodo visitado	Q
1.	1	2, 3, 4
2.	2	3, 4, 5, 6
3.	3	4, 5, 6
4.	4	5, 6, 7, 8
5.	5	6, 7, 8
6.	6	7, 8
7.	7	8
8.	8	—

Al igual que para el recorrido en profundidad, podemos asociar un árbol al recorrido en anchura. La figura 9.10 muestra el árbol generado por la búsqueda anterior. Las aristas del grafo que no tienen una arista correspondiente en el arbol se representan mediante líneas discontinuas; véase el problema 9.30. En general, si el grafo G que se está recorriendo no es conexo, el recorrido genera un bosque de árboles, uno por cada componente conexo de G.

Resulta sencillo mostrar que el tiempo requerido por un recorrido en anchura es del mismo orden que el requerido por un recorrido en profundidad, a saber, $\Theta(\text{máx}(a,n))$. Si se emplea la interpretación adecuada de la palabra «adyacente», el algoritmo de recorrido en anchura —una vez más, exactamente igual que el algoritmo de recorrido en profundidad— se puede aplicar sin modificación tanto a los grafos dirigidos como a los no dirigidos; véanse los problemas 9.31 y 9.32.

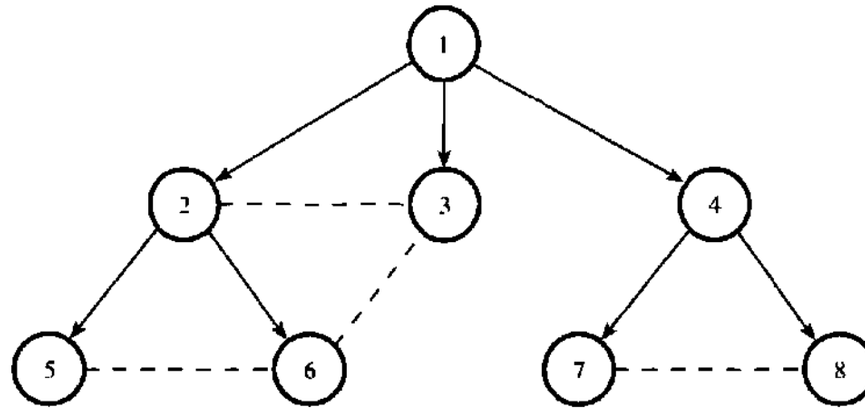


Figura 9.10. Un árbol de recorrido en anchura

El recorrido en anchura se emplea especialmente cuando haya que efectuar una exploración parcial de un grafo infinito (o tan grande que no resulte manejable), o para hallar el camino más corto desde un punto de un grafo hasta otro. Consideremos por ejemplo el problema siguiente. Se nos da el valor 1. Para construir otros valores, disponemos de dos operaciones: la multiplicación por 2 y la división por 3. Para la segunda operación, el operando debe ser mayor que 2 (por lo que no podemos llegar al 0), y se descartan los posibles decimales. Si se ejecutan las operaciones de izquierda a derecha, se puede obtener el valor 10 de la forma siguiente:

$$10 = 1 \times 2 \times 2 \times 2 \times 2 \div 3 \times 2$$

Deseamos obtener un valor concreto n. ¿Cómo podríamos hacerlo?

El problema se puede representar como una búsqueda en el grafo dirigido infinito de la figura 9.11. Aquí el valor dado 1 se encuentra en el nodo de la posición superior izquierda. A partir de ahí, cada nodo se une con los valores que se pueden obtener empleando las dos operaciones disponibles. Por ejemplo, a partir del valor 16 podemos obtener los dos valores nuevos 32 (multiplicando por 2) y 5 (dividiendo 16 por 3 y descartando los decimales). Por claridad, hemos omitido los enlaces que retroceden hacia valores que ya están disponibles; por ejemplo desde 8 hasta 2. Sin embargo, los enlaces hacia atrás están presentes en el grafo real. El grafo es infinito, porque una secuencia tal como 1, 2,..., 512,... puede continuar indefinidamente. No se trata de un árbol, porque el nodo 42, por ejemplo, se puede alcanzar tanto desde 128 como desde 21. Cuando se incluyen los enlaces hacia atrás, ni siquiera es acíclico.

Para resolver un caso dado del problema, esto es, la forma de construir un valor concreto de n, buscaremos en el grafo empezando por uno hasta encontrar el valor que estemos buscando. En este grafo infinito, sin embargo, un recorrido en profundidad puede no funcionar. Supongamos por ejemplo que $n = 13$. Si exploramos los vecinos de los nodos siguiendo el orden «primero multiplicar por 2, luego dividir por 3», entonces la búsqueda en profundidad va visitando sucesivamente los nodos 1, 2, 4,..., y así sucesivamente, y se va por la rama superior y (dado que siempre hay un nuevo vecino que examinar) nunca encuentra una situación que le obligue a retroceder a un nodo anterior. En este caso, la búsqueda fallará con certeza. Por otra parte, si explora-

mos los vecinos de un nodo por el orden «primero división por 3, después multiplicación por 2», entonces la búsqueda baja primero al nodo 12; desde allí avanza sucesivamente hasta los nodos 24, 48, 96, 32 y 64, y desde el 64 se pierde en la parte superior derecha del grafo. Quizá tengamos la suerte de volver al nodo 13, hallando así la solución de nuestro problema, pero nada lo garantiza. Aun cuando la encontrásemos, la solución hallada sería más compleja de lo necesario. Si se programa este recorrido en profundidad en una computadora, se encontrará de hecho que se alcanza el valor dado 13 al cabo de 74 operaciones de multiplicación y división.

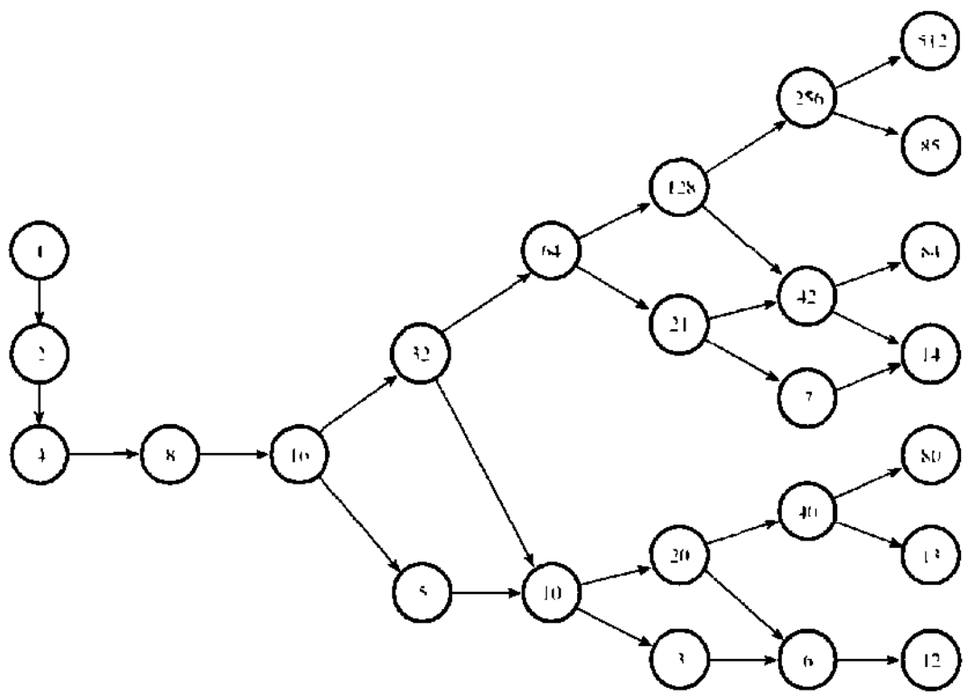


Figura 9.11. Multiplicación por 2 y división por 3

Por otra parte, una búsqueda en anchura tiene la certeza de hallar una solución si existe. Si examinamos los vecinos por el orden «primero multiplicar por 2, luego dividir por 3», una búsqueda en anchura que comience en uno visitará los nodos 1, 2, ..., 16, 32, 5, 64, 10, 128, 21, 20, 3, y así sucesivamente. No sólo estamos seguros de hallar el valor buscado si está en el grafo, sino que además la solución obtenida empleará el menor número posible de operaciones. En otras palabras, el camino hallado desde 1 hasta el valor deseado n será lo más corto posible. Lo mismo sucede si efectuamos la búsqueda examinando los vecinos por el orden «primero dividir por 3, luego multiplicar por 2». Utilizando cualquiera de estas dos búsquedas en anchura, resulta sencillo (incluso a mano) descubrir varias maneras de producir el valor 13 empleando solamente 9 operaciones. Por ejemplo

$$13 = 1 \times 2 \times 2 \times 2 \times 2 \div 3 \times 2 \times 2 \times 2 \div 3$$

Por supuesto, incluso una búsqueda en anchura puede no tener éxito. En nuestro ejemplo, puede ser que algunos valores de n no estén ni siquiera presentes en el grafo. (Desconociendo si esto es o no verdad, dejaremos esta cuestión como ejercicio para el lector.) En este caso, cualquier técnica de búsqueda

fallaría para los valores que no están presentes. Si un grafo incluye uno o más nodos con un número infinito de vecinos, pero sin caminos de longitud infinita, la búsqueda en profundidad puede tener éxito aun cuando falle el recorrido en anchura. Sin embargo, esta situación parece producirse con menos frecuencia que la inversa.

9.6 VUELTA ATRÁS

Tal como vimos anteriormente, hay distintos problemas que se pueden concebir en términos de grafos abstractos. Por ejemplo, veíamos en la Sección 9.1 que se pueden utilizar los nodos de un grafo para representar las situaciones de un juego de ajedrez, y las aristas para representar jugadas válidas. Con frecuencia, el problema original se traduce a buscar un nodo, un camino o un patrón específicos en el grafo asociado. Si el grafo contiene un número elevado de nodos, y especialmente si es infinito, puede resultar inútil o inabordable construirlo explícitamente en la memoria de una computadora antes de aplicar una de las técnicas de búsqueda que hemos empleado hasta el momento.

En tales situaciones utilizaremos un *grafo implícito*. Éste es un grafo para el cual se dispone de una descripción de sus nodos y aristas, de tal manera que se pueden construir partes relevantes del grafo a medida que progresa el recorrido. De esta manera, se ahorra tiempo de computación siempre que el recorrido tenga éxito antes de haber construido todo el grafo. La economía de espacio en memoria también puede ser dramática, sobre todo cuando se pueden descartar los nodos que hayan sido examinados ya, creando así espacio para poder explorar los nodos subsiguientes. Si el grafo en cuestión es infinito, las técnicas como ésta constituyen nuestra única esperanza de llegar a explorarlo. Esta sección, y las que siguen, detallan algunas formas estándar de organizar recorridos en un grafo implícito.

En su forma básica, la vuelta atrás se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos. Sea cual sea su estructura, existe sólo implícitamente. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas). Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.

9.6.1 El problema de la mochila (3)

Como primer ejemplo para ilustrar el principio general, volvamos al problema de la mochila que se describía en la Sección 8.4. Recordemos que se nos dan un cierto número de objetos y una mochila. En esta ocasión, sin embargo, en lugar de suponer que están disponibles n objetos, supondremos que lo que tenemos son n *tipos* de objetos, y que está disponible un número adecuado de objetos de cada tipo. Esto no altera el problema de forma sustancial. Para $i = 1, 2, \ldots, n$ un objeto de tipo i tiene un peso positivo w_i y un valor positivo v_i. La mochila puede llevar un peso que no exceda de W. Nuestro objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos incluidos, respetando al mismo tiempo la restricción de capacidad. Podemos tomar un objeto o prescindir de él, pero no podemos tomar una cierta parte de un objeto.

Supongamos para concretar que deseamos resolver un caso del problema con cuatro tipos de objetos, cuyos pesos son respectivamente 2, 3, 4 y 5 unidades, y cuyos valores son 3, 5, 6 y 10 respectivamente. La mochila puede llevar un máximo de 8 unidades de peso. Esto se puede hacer utilizando la técnica de vuelta atrás, explorando el árbol implícito de la figura 9.12.

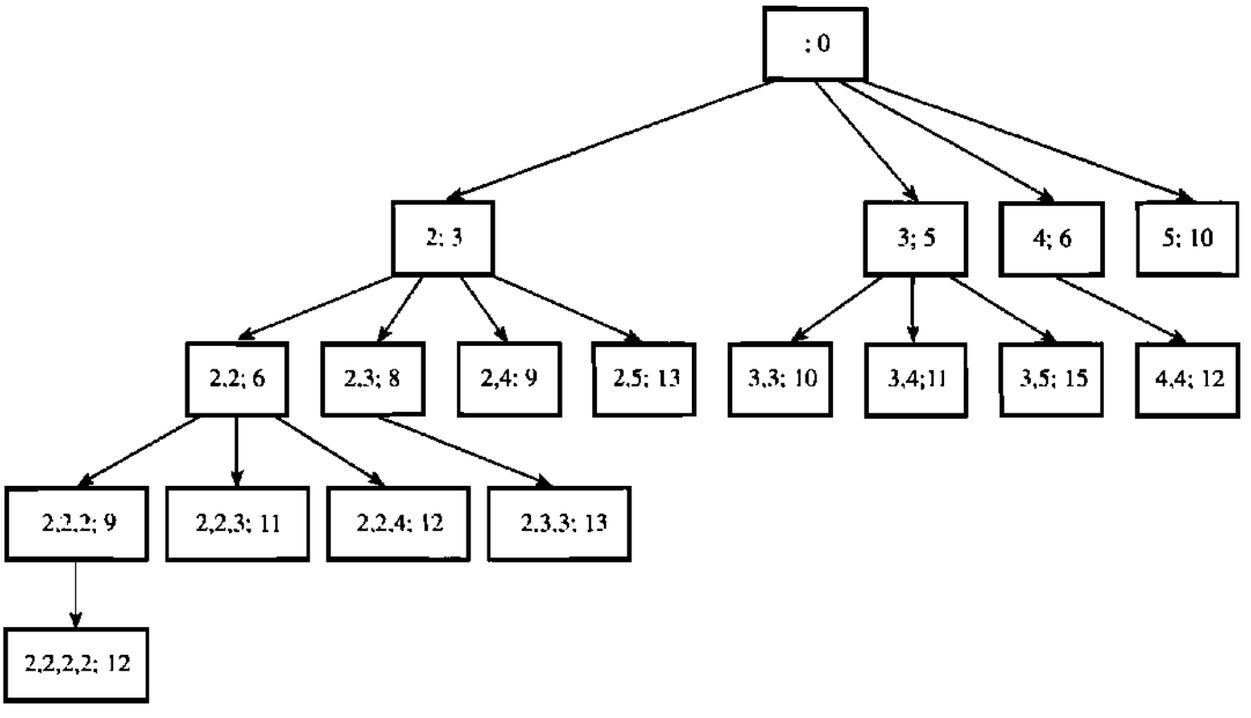


Figura 9.12. El árbol implícito para un problema de mochila

Aquí, un nodo tal como el (2, 3; 8) corresponde a una solución parcial de nuestro problema. Las cifras que están a la izquierda del punto y coma son los pesos de los objetos que hemos decidido incluir, y la cifra que está a la derecha es el valor actual de la carga. Bajar de un nodo a uno de sus hijos se corresponde con decidir qué clase de objeto se va a poner en la mochila a continuación. Sin pérdida de generalidad, podemos acordar que cargaremos los objetos en la mochila por orden creciente de peso. Esto no es esencial, y a decir verdad cualquier otro orden —por orden decre-

ciente de pesos, por ejemplo, o por valores— funcionaría igual de bien, pero así se reduce el tamaño del árbol que hay que explorar. Por ejemplo, una vez visitado el nodo (2, 2, 3; 11) no tiene sentido visitar posteriormente el (2, 3, 2; 11).

Inicialmente, la solución parcial está vacía. El algoritmo de vuelta atrás explora el árbol como un recorrido en profundidad, construyendo nodos y soluciones parciales a medida que avanza. En el ejemplo, el primer nodo que se visita es el (2; 3), el siguiente es el (2, 2; 6), el tercero es (2, 2, 2; 9) y el cuarto es (2, 2, 2, 2; 12). A medida que se visita cada nodo nuevo, se extiende la solución parcial. Después de visitar estos cuatro nodos, se bloquea el recorrido en profundidad: el nodo (2, 2, 2, 2; 12) no posee sucesores no visitados (de hecho, carece de sucesores), puesto que añadir más elementos a esta solución parcial violaría la restricción de capacidad. Dado que esta solución parcial puede resultar ser la solución óptima de nuestro caso, la memorizamos.

El algoritmo de recorrido en profundidad vuelve atrás ahora en busca de otras soluciones. En cada paso retrocedido por el árbol, se elimina el elemento correspondiente de la solución parcial. En el ejemplo, el recorrido vuelve primero a (2, 2, 2; 9), que también carece de sucesores no visitados; sin embargo, al retroceder un paso más por el árbol hasta el nodo (2, 2; 6), hay dos sucesores que quedan por visitar. Después de explorar los nodos (2, 2, 3; 11) y (2, 2, 4; 12), ninguno de los cuales mejora la solución memorizada anteriormente, la búsqueda retrocede otro paso más, y así sucesivamente. Explorando el árbol de esta manera, encontramos que (2, 3, 3; 13) es una solución mejor que la que ya tenemos, y después (3, 5; 15) es todavía mejor. Puesto que no se hace ninguna otra mejora antes de finalizar la búsqueda, ésta es la solución óptima del caso.

Programar el algoritmo es sencillo e ilustra la íntima relación existente entre recursión y recorrido en profundidad. Supongamos que los valores de n y de W, y los valores de la matrices $w[1..n]$ y de $v[1..n]$ correspondientes al caso que hay que resolver están disponibles como variables globales. La ordenación de los tipos de elementos es irrelevante. Se define una función *mochilava* de la forma siguiente:

```
función mochilava(i, r)
    {Calcula el valor de la mejor carga que se puede
    construir empleando elementos de los tipos i a n
    y cuyo peso total no sobrepase r}
    b ← 0
    {Se prueban por turno las clase de objetos admisibles }
    para k ← i hasta n hacer
        si w[k] ≤ r entonces
        b ← máx(b, v[k] + mochilava(k, r−w[k]))
    devolver b
```

Ahora, para hallar el valor de la mejor carga, llamamos a *mochilava*(1, W). Cada llamada recursiva a *mochilava* se corresponde con extender el recorrido en profundidad hasta el nivel inmediatamente inferior del árbol, mientras que el bucle **para** se encarga de examinar todas las posibilidades en un nivel dado. En esta ver-

sión del programa, la composición de la carga que se está examinando está dada implícitamente por los valores de k en la pila recursiva. No es difícil adaptar el programa para que nos dé explícitamente la composición de la mejor carga junto con su valor; véase el Problema 9.42.

9.6.2 El problema de las ocho reinas

Para nuestro segundo ejemplo de vuelta atrás, consideremos el problema clásico consistente en situar las ocho reinas en un tablero de ajedrez de tal modo que ninguna de ellas amenace a ninguna de las demás. Recordemos que una reina amenaza a los cuadrados de la misma fila, de la misma columna o de las mismas diagonales.

La forma más evidente de resolver este problema consiste en probar sistemáticamente todas las formas de colocar ocho reinas en un tablero, determinando en cada ocasión si se ha encontrado una solución. Esta aproximación no tiene aplicación práctica, ni siquiera con una computadora, porque el número de situaciones que habría que examinar es $\binom{64}{8} = 4.426.165.368$. La primera mejora que podemos considerar consiste en no poner nunca más de una reina en una fila. Esto reduce la representación del tablero a un vector de ocho elementos, cada uno de los cuales da la posición de la reina dentro de la fila correspondiente. Por ejemplo, el vector (3, 1, 6, 2, 8, 6, 4, 7) representa la situación en que la reina de la fila 1 está en la columna 3, la reina de la fila 2 está en la columna 1, y así sucesivamente. Esta situación concreta no es una solución para nuestro problema, puesto que las reinas de las filas 3 y 6 se encuentran en la misma columna, y además hay dos pares de reinas en la misma diagonal. Empleando esta representación, podríamos escribir un algoritmo empleando ocho bucles anidados (para la función *solución*, véase el Problema 9.43):

```
programa reinas1
    para i1 ← 1 hasta 8 hacer
        para i2 ← 1 hasta 8 hacer
            ...
                para i8 ← 1 hasta 8 hacer
                    sol ← [i1, i2,...i8]
                    si solución(sol) entonces escribir sol
                                             parar
    escribir «no hay solución»
```

El número de situaciones que hay que considerar se ha reducido a $8^8 = 16.777.216$, aunque de hecho el algoritmo se detiene después de considerar solamente 1.299.852 situaciones.

La representación del tablero mediante un vector nos impide intentar poner dos reinas en una misma fila. Una vez que nos damos cuenta de esto, es natural ser igualmente sistemáticos en lo tocante a las columnas. Por tanto, ahora representaremos el tablero mediante un vector formado por ocho números *diferentes* entre 1 y 8, esto es, mediante una permutación de los ocho primeros números enteros. Esto produce el siguiente algoritmo:

```
programa reinas2
    sol ← permutación inicial
    mientras sol ≠ permutación final y no solución(sol) hacer
            sol ← permutación siguiente
    si solución(sol) entonces escribir sol
                          sino escribir «no hay solución»
```

Hay varias maneras naturales de generar sistemáticamente todas las permutaciones de los n primeros números enteros. Por ejemplo, podemos poner cada valor correspondiente en la primera posición, y generar recursivamente, para cada valor inicial, todas las permutaciones de los $n-1$ elementos restantes. El procedimiento siguiente muestra la manera de hacer esto. Aquí $T[1..n]$ es una matriz global a la que se da como valor inicial [1, 2,..., n], y la llamada inicial al procedimiento es *perm*(1). Esta forma de generar permutaciones es también una especie de vuelta atrás.

```
procedimiento perm(i)
    si i = n entonces usar(T) {T es una nueva permutación}
            sino para j ← i hasta n hacer intercambiar T[i] y T[j]
                                          perm(i+1)
                                          intercambiar T[i] y T[j]
```

Este enfoque reduce el número de situaciones posibles a 8! = 40.320. Si se utiliza el algoritmo precedente para generar las permutaciones, entonces en realidad solo se consideran 2.830 situaciones antes de que el algoritmo encuentre una solución. Es más complicado generar las permutaciones que generar todos los posibles vectores de ocho enteros entre 1 y 8. Por otra parte, resulta más sencillo verificar en este caso si una situación dada es una solución. Dado que ya sabemos que dos reinas no pueden estar ni en la misma fila ni en la misma columna, basta verificar que no estén en la misma diagonal.

Partiendo de un método muy grosero que ponía reinas absolutamente en todos los lugares del tablero, hemos progresado primero hasta un método que nunca pone dos reinas en la misma fila, y después a otro mejor que sólo considera aquellas situaciones en que dos reinas nunca pueden estar ni en la misma fila ni en la misma columna. Sin embargo, todos estos algoritmos comparten un defecto común: nunca comprueban si una situación es una solución mientras no se hayan colocado todas las reinas en el tablero. Por ejemplo, incluso el mejor de ellos hace 720 intentos inútiles para colocar las seis últimas reinas en el tablero cuando ha comenzado por poner las dos primeras en la diagonal principal, lugar en que desde luego se amenazan una a otra.

La vuelta atrás nos permite mejorar esto. Como primer paso, reformulamos el problema de las ocho reinas como un problema de búsqueda en un árbol. Decimos que un vector $V[1..k]$ de enteros entre 1 y 8 es *k-prometedor*, para $0 \leq k \leq 8$, si ninguna de las k reinas colocadas en las posiciones (1, $V[1]$),(2, $V[2]$),...,(k, $V[k]$) amenaza a ninguna de las otras. Matemáticamente, un vector V es *k-prometedor* si, para todo par de enteros i y j entre 1 y k, con $i \neq j$, tenemos que $V[i]-V[j] \notin \{i-j, 0, j-i\}$. Pa-

ra $k \leq 1$, todo vector V es *k-prometedor*. Las soluciones del problema de las 8 reinas se corresponden con aquellos vectores que son 8-prometedores.

Sea N el conjunto de vectores k-prometedores, $0 \leq k \leq 8$. Sea $G = \langle N, A \rangle$ el grafo dirigido tal que $(U, V) \in A$ si y sólo si existe un entero k, con $0 \leq k < 8$, tal que

- U es k-prometedor
- V es $(k + 1)$-prometedor, y
- $U[i] = V[i]$ para todo $i \in [1..k]$

Este grafo es un árbol. Su raíz es el vector vacío correspondiente a $k = 0$. Sus hojas son o bien soluciones ($k = 8$) o posiciones sin salida ($k < 8$) tales como [1, 4, 2, 5, 8]: en tal situación, resulta imposible colocar una reina en la fila siguiente sin amenazar por lo menos a una de las reinas que ya están en el tablero. Las soluciones del problema de las ocho reinas se pueden obtener explorando este árbol. Sin embargo, no generamos explícitamente el árbol para explorarlo después. Más bien, se generan y abandonan los nodos en el transcurso de la exploración. El recorrido en profundidad es el método evidente, sobre todo si sólo necesitamos una solución.

Esta técnica posee dos ventajas con respecto al algoritmo que va probando sistemáticamente todas las permutaciones. En primer lugar, el número de nodos del árbol es menor que 8! = 40.320. Aunque no resulta sencillo calcular teóricamente este número, es fácil contar los nodos empleando una computadora: hay 2.057. De hecho, basta con explorar 114 nodos para obtener una primera solución. En segundo lugar, para decidir si un vector es k-prometedor, sabiendo que es una extensión de un vector $(k-1)$-prometedor, sólo necesitamos comprobar la última reina que haya que añadir. Esto se puede acelerar si asociamos a cada nodo prometedor el conjunto de columnas, el de diagonales positivas (a 45 grados) y el de diagonales negativas (a 135 grados) controlados por las reinas que ya están puestas.

En el procedimiento siguiente, *sol*[1..8] es una matriz global. Para imprimir todas las soluciones del problema de las ocho reinas, se llama a *reinas*(0, Ø, Ø, Ø):

```
procedimiento reinas(k, col, diag45, diag135)
    {sol[1..k] es k-prometedor,
    col = {sol[i] | 1 ≤ i ≤ k},
    diag45 = {sol[i] − i + 1 | 1 ≤ i ≤ k} y
    diag135 = {sol[i] + i − 1 | 1 ≤ i ≤ k} }
  si k = 8 entonces {un vector 8-prometedor es una solución}
                    escribir sol
  sino {explorar las extensiones (k+1)-prometedoras de sol}
    para j ← 1 hasta 8 hacer
      si j ∉ col y j - k ∉ diag45 y j + k ∉ diag135
      entonces sol[k + 1] ← j
        {sol[1..k+1] es (k+1)-prometedor}
        reinas(k+1, col ∪ {j},
          diag45 ∪ {j - k}, diag135 ∪ {j + k})
```

Está claro que el problema se puede generalizar a un número arbitrario de reinas: ¿cómo se pueden situar n reinas en un «tablero» $n \times n$ de tal manera que ninguna de ellas amenace a ninguna de las demás? Tal como cabe esperar, la ventaja obtenida al utilizar la vuelta atrás en lugar de un enfoque exhaustivo se vuelve más pronunciada a medida que crece n. Por ejemplo, para $n = 12$ son 479.001.600 las posibles permutaciones que hay que considerar. Empleando el generador de permutaciones dado anteriormente, la primera solución que se encuentra corresponde a la 4.546.044-ésima situación examinada. Por otra parte, el árbol explorado por el algoritmo de vuelta atrás contiene solamente 856.189 nodos, y se obtiene una solución al visitar el nodo 262. El problema se puede generalizar todavía más colocando las «reinas» en tres dimensiones en un tablero $n \times n \times n$; véase el Problema 9.49.

9.6.3 El caso general

Los algoritmos de vuelta atrás se pueden utilizar aun cuando las soluciones buscadas no tengan todas necesariamente la misma longitud. Véase el esquema general:

```
procedimiento vueltaatrás(v[1..k])
        {v es un vector k-prometedor}
        si v es una solución entonces escribir v
        sino para cada vector (k+1)-prometedor w
                        tal que w[1..k] = v[1..k]
                hacer vueltaatrás(w[1..k+1])
```

El **sino** debería estar presente si y sólo si resulta imposible disponer de dos soluciones diferentes tales que una sea prefijo de la otra.

Tanto el problema de la mochila como el problema de las n reinas se resolvían empleando una búsqueda en profundidad en el árbol correspondiente. Algunos problemas que se pueden formular en términos de explorar un grafo implícito tienen la propiedad consistente en que corresponden a un grafo infinito. En este caso puede ser necesario emplear un recorrido en anchura para evitar la exploración interminable de alguna rama infinita carente de resultados positivos. El recorrido en anchura también es apropiado si tenemos que encontrar una solución comenzando a partir de una cierta situación inicial y efectuando el menor número de pasos posible.

9.7 RAMIFICACIÓN Y PODA

Al igual que la vuelta atrás, la ramificación y poda es una técnica para explorar un grafo dirigido implícito. Una vez más, este grafo es normalmente acíclico, o incluso un árbol. Esta vez vamos a buscar la solución óptima de algún problema. En cada nodo, calculamos una cota del posible valor de aquellas soluciones que pudieran encontrarse más adelante en el grafo. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento, entonces no necesitamos seguir explorando esta parte del grafo.

En su versión más sencilla, el cálculo de cotas se combina con un recorrido en anchura o en profundidad, y solamente sirve para podar ciertas ramas de un árbol o para cerrar caminos de un grafo. Con más frecuencia, sin embargo, la cota calculada se utiliza también para seleccionar el camino que, entre los abiertos, parezca más prometedor para explorarlo primero.

En términos generales, podemos decir que las búsquedas en profundidad acaban de explorar los nodos por orden inverso al de su creación, empleando una pila para almacenar los nodos que se han generado pero no han sido examinados aún. Las búsquedas en amplitud terminan de explorar los nodos por el mismo orden en que son creados, utilizando una cola para almacenar los nodos que se han generado pero no han sido examinados aún. Ramificación y poda utiliza cálculos auxiliares para decidir en cada momento qué nodo debe explorarse a continuación y una lista con prioridad para almacenar los nodos que se han generado pero no han sido examinados aún. Recordemos que los montículos suelen ser ideales para almacenar listas con prioridad; véase la Sección 5.7. Ilustraremos la técnica con dos ejemplos.

9.7.1 El problema de la asignación

En el *problema de la asignación*, hay que asignar n tareas a n agentes, de forma que cada agente realice exactamente una tarear. Si al agente i, con $1 \leq i \leq n$, se le asigna la tarea j, con $1 \leq j \leq n$, entonces el coste de realizar esta tarea concreta será c_{ij}. Dada la matriz de costes completa, el problema consiste en asignar tareas a los agentes de tal manera que se minimice el coste total de ejecutar las n tareas.

Por ejemplo, consideremos tres agentes a, b y c, a los que hay que asignar las tareas 1, 2 y 3 con la matriz de costes siguiente:

	1	2	3
a	4	7	3
b	2	6	1
c	3	9	4

Si asignamos la tarea 1 al agente a, la tarea 2 al agente b y la tarea 3 al agente c, entonces nuestro coste total será $4 + 6 + 4 = 14$, mientras que si asignamos la tarea 3 al agente a, la tarea 2 al agente b y la tarea 1 al agente c, el coste será solamente $3 + 6 + 3 = 12$. En este caso concreto, el lector puede verificar que la asignación óptima es $a \rightarrow 2$, $b \rightarrow 3$ y $c \rightarrow 1$, cuyo coste es $7 + 1 + 3 = 11$.

El problema de la asignación tiene numerosas aplicaciones. Por ejemplo, en lugar de hablar de agentes y tareas, podemos formular el problema en términos de edificios y solares, donde c_{ij} es el coste de construir el edificio i en el solar j, y deseamos minimizar el coste total de los edificios. Es fácil inventar otros ejemplos. En general, con n agentes y n tareas, hay $n!$ posibles asignaciones para considerar que son demasiadas incluso para valores moderados de n. Por tanto, recurriremos a la ramificación y poda.

Supongamos que es preciso resolver el caso cuya matriz de costes se muestra en la figura 9.13. Para obtener una cota superior de la respuesta, obsérvese que $a \rightarrow 1$, $b \rightarrow 2$, $c \rightarrow 3$, $d \rightarrow 4$ es una posible solución cuyo coste es 11 + 15 + 19 + 28 = 73. La solución óptima del problema no puede costar más que esto. Otra posible solución es $a \rightarrow 4$, $b \rightarrow 3$, $c \rightarrow 2$, $d \rightarrow 1$, cuyo coste se obtiene sumando los elementos de la otra diagonal de la matriz de costes, dando 40 + 13 +17 + 17 = 87. En este caso, la segunda solución no supone mejora respecto a la primera. Para obtener una cota inferior de la solución, podemos argumentar que sea quien sea el que ejecute la tarea 1, el coste será 11 como mínimo; sea quien sea el que ejecute la tarea 2, el coste será de 12 como mínimo, y así sucesivamente. Por tanto, sumando los elementos más pequeños de cada columna obtenemos una cota inferior de la respuesta. En el ejemplo, esto es 11 + 12 + 13 + 22 = 58. Se obtiene una segunda cota inferior sumando los elementos más pequeños de cada fila, basándonos en que todos los agentes tienen que hacer algo. En este caso obtenemos 11 + 13 +11 + 14 = 49, que es menos útil que la cota inferior anterior. Juntando estos resultados, sabemos que la respuesta de nuestro caso se encuentra en algún lugar de [58..73].

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Figura 9.13. La matriz de costes para el problema de asignación

Para resolver el problema mediante ramificación y poda, exploraremos un árbol cuyos nodos corresponden a asignaciones parciales. En la raíz del árbol no se han hecho asignaciones. En lo sucesivo, en cada nivel se determina la asignación de un agente más. Para cada nodo, calculamos una cota de las soluciones que se pueden obtener completando la asignación parcial correspondiente, y utilizamos esta cota para cerrar caminos y para guiar la búsqueda. Supongamos por ejemplo que empezando por la raíz decidimos efectuar en primer lugar la asignación del agente *a*. Dado que hay cuatro formas de hacer esto, hay cuatro ramas que parten de la raíz. La figura 9.14 ilustra la situación.

La cifra que aparece junto a cada nodo es una cota inferior de las soluciones que se pueden obtener completando la asignación parcial correspondiente. Ya hemos mostrado la forma en que se puede obtener la cota 58 en la raíz. Para calcular la cota del nodo $a \rightarrow 1$, por ejemplo, observemos en primer lugar que con esta asignación parcial la tarea 1 tendrá un coste 11. La tarea 2 será ejecutada por *b*, *c* o *d* así

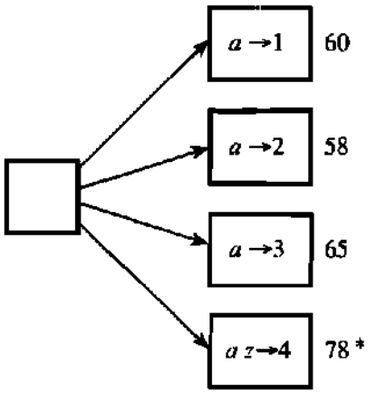


Figura 9.14. Después de asignar una tarea al agente a

que el mínimo coste posible es 14. De forma similar, las tareas 3 y 4 también serán ejecutadas por *b*, *c* o *d*, y sus costes mínimos posibles son 13 y 22 respectivamente. De esta manera, una cota inferior de cualquier solución que se obtenga completando la asignación parcial $a \rightarrow 1$ es 11 + 14 + 13 + 22 = 60. De forma similar, para el nodo $a \rightarrow 2$, la tarea 2 será ejecutada por el agente *a* con un coste 12, mientras que las tareas 1, 3 y 4 serán ejecutadas por los agentes *b*, *c* y *d* con un coste mínimo de 11, 13 y 22 respectivamente. Por tanto, toda solución que incluya la asignación $a \rightarrow 2$ costará como mínimo 12 + 11 + 13 + 22 = 58. Las otras dos cotas inferiores se obtienen de forma similar. Dado que sabemos que la solución óptima no puede sobrepasar 73, ya está claro que no tiene sentido seguir explorando el nodo $a \rightarrow 4$: toda solución obtenida completando esta asignación parcial costará al menos 78, así que no puede ser óptima. El asterisco de este nodo denota que está «muerto». Sin embargo, los otros tres nodos siguen estando vivos. El nodo $a \rightarrow 2$ tiene la cota inferior más pequeña. Argumentando que por tanto parece más prometedor que los otros, será éste el que exploremos a continuación. Hacemos esto fijando un elemento más de la asignación parcial, digamos el *b*. De esta manera llegamos a la situación que se muestra en la figura 9.15.

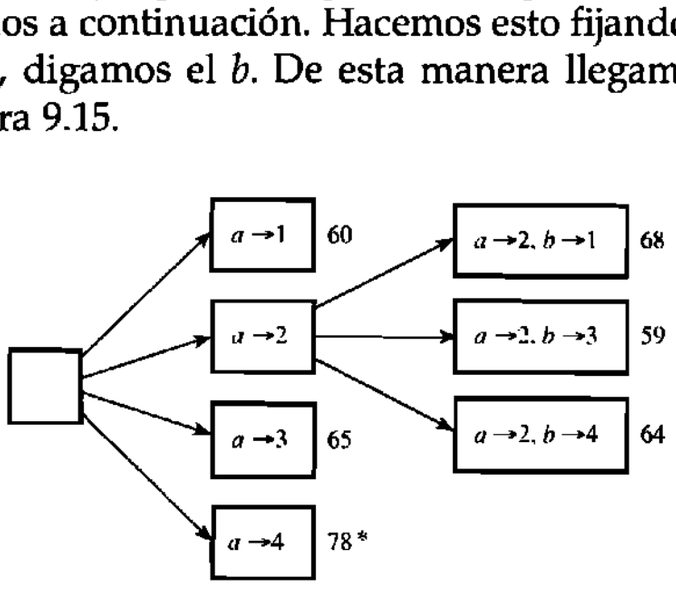


Figura 9.15. Después de asignar una tarea al agente b

Una vez más, la cifra que aparece junto a cada nodo nos da una cota inferior del coste de las soluciones que se pueden obtener completando la asignación parcial

correspondiente. Por ejemplo, en el nodo $a \rightarrow 2, b \rightarrow 1$, la tarea 1 costará 14 y la tarea 2 costará 12. Las tareas restantes 3 y 4 deben ser ejecutadas por c o d. El coste más pequeño posible para la tarea 3 es por tanto 19, mientras que para la tarea 4 es 23. Por tanto, una cota inferior de las soluciones posibles es 14 + 12 + 19 + 23 = 68. Las otras dos cotas nuevas se calculan de forma similar.

El nodo más prometedor del árbol es ahora $a \rightarrow 2$, $b \rightarrow 3$ con una cota inferior de 59. Para seguir explorando el árbol comenzando en este nodo, fijamos un elemento más de la asignación parcial, digamos el c. Cuando se fijan las asignaciones de a, b y c, sin embargo, ya no nos queda opción alguna para la asignación de d, así que la solución está completa. Los nodos de la derecha de la figura 9.16, que muestra la siguiente fase de nuestra exploración, corresponden por tanto a soluciones completas.

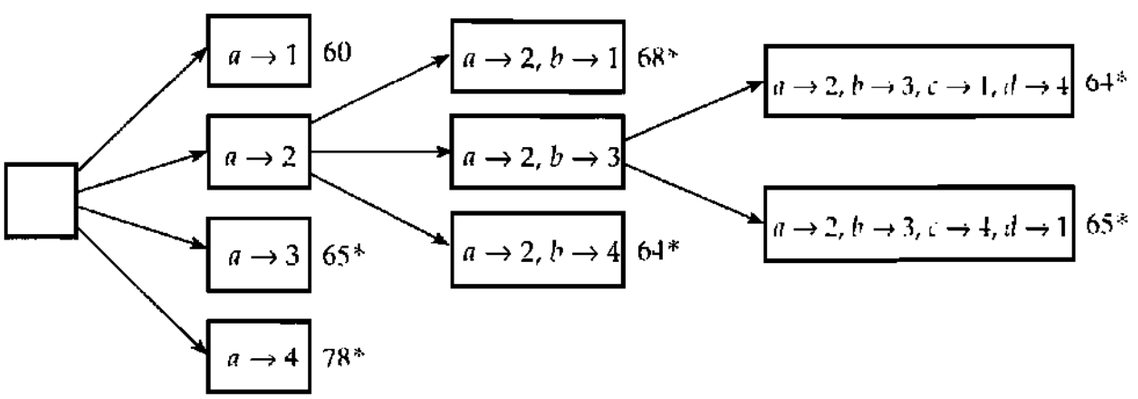


Figura 9.16. Después de asignar una tarea al agente c

La solución $a \rightarrow 2, b \rightarrow 3, c \rightarrow 1, d \rightarrow 4$, con un coste de 64, es mejor que cualquiera de las soluciones halladas al empezar, y nos proporciona una nueva cota superior para el óptimo. Gracias a esta nueva cota superior, podemos descartar los nodos $a \rightarrow 3$ y $a \rightarrow 2, b \rightarrow 1$ a efectos de posteriores exploraciones, tal como indican los asteriscos. Ninguna solución que complete estas asignaciones parciales puede ser tan buena como la que acabamos de encontrar. Si solamente deseamos una solución del caso, podemos eliminar también el nodo $a \rightarrow 2, b \rightarrow 4$.

El único nodo que sigue mereciendo la pena explorar es $a \rightarrow 1$. Procediendo igual que antes, al cabo de dos pasos obtenemos la situación final que se muestra en la figura 9.17. La mejor solución hallada es $a \rightarrow 1, b \rightarrow 3, c \rightarrow 4$ y $d \rightarrow 2$, con un coste de 61. En los nodos restantes no explorados, la cota inferior es mayor que 61, así que no tiene sentido seguir estudiándolos. La solución anterior es por tanto la solución óptima de nuestro caso.

El ejemplo ilustra que aun cuando en una fase anterior el nodo $a \rightarrow 2$ era el más prometedor, la solución óptima no llegó a surgir de él. Para obtener nuestra respuesta, hemos construido 15 de los 41 nodos (1 raíz, 4 de profundidad 1, 12 de profundidad 2, y 24 de profundidad 3) que están presentes en un árbol completo del tipo ilustrado. De las 24 soluciones posibles, sólo 6 (incluyendo las dos utilizadas para determinar la cota superior inicial) han sido examinadas.

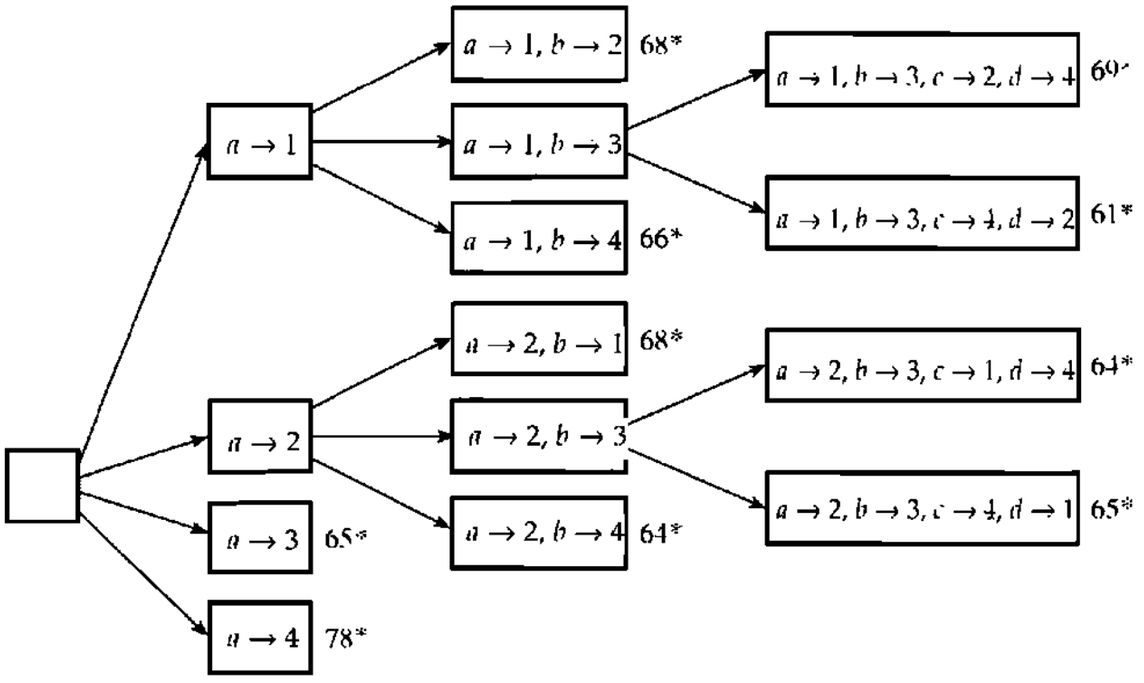


Figura 9.17. El árbol explorado en su totalidad

9.7.2 El problema de la mochila (4)

Como segundo ejemplo, consideremos el problema de la mochila; véanse las Secciones 8.4. y 9.6.1. Aquí se nos pide maximizar $\sum_{i=1}^{n} x_i v_i$ sometido a la restricción $\sum_{i=1}^{n} x_i w_i \leq W$, donde los v_i y w_i son todos estrictamente positivos, y los x_i son enteros no negativos. Este problema también se puede resolver por ramificación y poda.

Supongamos sin pérdida de generalidad que las variables están numeradas de tal manera que $v_i / w_i \geq v_{i+1} / w_{i+1}$. Entonces si los valores de $x_1, x_2, \ldots, x_k, 0 \leq k \leq n$ quedan fijados, con $\sum_{i=1}^{n} x_i w_i \leq W$, es fácil ver que el valor que se puede obtener sumando más elementos de los tipos $k+1,\ldots,n$ a la mochila no puede sobrepasar el valor

$$\sum_{i=1}^{k} x_i v_i + \left(W - \sum_{i=1}^{k} x_i w_i \right) v_{k+1} / w_{k+1}$$

Donde el primer término da el valor de los elementos que ya están en la mochila, y el segundo es una cota del valor que se puede añadir.

Para resolver el problema por ramificación y poda, exploramos un árbol en cuya raíz no está fijado el valor de ninguno de los x_i, y en cada nivel sucesivo se va determinando el valor de una variable más, por orden numérico de variables. En cada nodo que exploremos, sólo generamos aquellos sucesores que satisfagan la restricción de peso, de tal manera que cada nodo tiene un número finito de sucesores. Siempre que se genera un nodo, calculamos una cota superior del valor de

la solución que se puede obtener completando la carga parcialmente especificada, y utilizamos estas cotas superiores para cortar ramas inútiles y para guiar la exploración del árbol. Dejamos los detalles para el lector.

9.7.3 Consideraciones generales

La necesidad de mantener una lista de nodos que han sido generados pero que no han sido explorados en su totalidad, situados en diferentes niveles del árbol y preferiblemente ordenados por orden de las cotas correspondientes, hace que la ramificación y poda resulten difíciles de programar. El montículo es una estructura de datos ideal para almacenar esta lista. A diferencia del recorrido en profundidad, y de las técnicas relacionadas, el programador no dispone de una formulación recursiva elegante de la ramificación y poda. Sin embargo, la técnica es tan potente que suele emplearse en aplicaciones prácticas.

Resulta casi imposible dar una idea precisa de lo bien que se va a comportar esta técnica en un problema dado, empleando una cota dada. Siempre hay que llegar a un compromiso en lo concerniente a la calidad de la cota calculada. Con una cota mejor, examinamos menos nodos, y si tenemos suerte se nos guiará a una solución óptima con más rapidez. Por otra parte, lo más probable es que pasemos más tiempo en cada nodo calculando la cota correspondiente. En el caso peor, puede ocurrir que incluso una cota excelente no nos permita cortar ninguna rama del árbol, así que se desperdiciará todo el trabajo adicional efectuado en cada nodo. En la práctica, sin embargo, para problemas del tamaño que suele encontrarse en las aplicaciones, casi siempre es rentable invertir el tiempo necesario para calcular la mejor cota posible (dentro de lo razonable). Por ejemplo, se encuentran aplicaciones tales como programación lineal y que se manejan mediante ramificación y poda, obteniéndose la cota en cada nodo mediante la resolución de un problema asociado de programación lineal, con variables continuas.

9.8 EL PRINCIPIO DE MINIMAX

Sea cual sea la técnica utilizada, persiste el molesto hecho de que para un juego tal como el ajedrez queda descartado un examen exhaustivo del grafo asociado. En esta situación, tenemos que conformarnos con una búsqueda parcial en torno a la situación actual. Éste es el principio que subyace a una importante heurística denominada *minimax*. Aunque esta heurística no nos permite asegurar que ganaremos, siempre que esto sea posible, halla una jugada de la cual puede esperarse razonablemente que se cuente entre las mejores jugadas disponibles, explorando solamente una parte del grafo a partir de alguna posición dada. La exploración del grafo se detiene normalmente antes de alcanzar las posiciones terminales, empleando uno entre varios criterios posibles, y las posiciones en que se detiene la exploración se evalúan de forma heurística. En cierto sentido, esto no es sino una versión sistemática del método utilizado por ciertos jugadores humanos, que consiste en examinar anticipadamente un pequeño número de jugadas. Nos limitaremos a esbozar la técnica.

Supongamos entonces que se desea jugar una buena partida al ajedrez. El primer paso es definir una función estática de evaluación *eval* que atribuya un valor a cada posible situación. Idealmente, el valor de *eval*(u) debería crecer a medida que la situación de u sea más favorable para las blancas. Se acostumbra a dar valores no muy distintos de cero a las situaciones en que ninguno de los jugadores tiene una ventaja definida, y grandes valores negativos a las situaciones que favorecen a las negras. Esta función de evaluación debe de tener en cuenta muchos factores: el número y tipo de las piezas restantes por ambas partes, el control del centro, la libertad de movimiento y demás. Es preciso llegar a un compromiso entre la precisión de la función y el tiempo que se necesita para calcularla. Cuando se aplique a una situación terminal, la función de evaluación debería devolver $+\infty$ si hay jaque mate contra las negras, $-\infty$ si hay jaque mate contra las blancas, y cero si la partida acaba en tablas. Por ejemplo, una función de evaluación que tiene en cuenta correctamente los aspectos estáticos de la situación, pero que resulta demasiado sencilla para utilizarla en la realidad, podría ser la siguiente: para situaciones no terminales, se cuenta 1 punto por cada peón blanco, $3\frac{1}{4}$ puntos por cada alfil o caballo blanco, 5 puntos por cada torre, y 10 puntos por cada reina blanca; se sustrae un número equivalente de puntos por cada pieza negra.

Si la función de evaluación estática fuera perfecta, sería sencillo determinar la mejor jugada que se puede hacer. Supongamos que es el turno de las blancas para mover en la posición u. El mejor movimiento sería pasar a la posición v que maximice *eval*(v) entre todos los sucesores de u:

```
val ← −∞
para cada posición w que sea un sucesor de u hacer
    si eval(w) ≥ val entonces val ← eval(w)
                              v ← w
```

Esta aproximación tan simple no tendría mucho éxito si empleáramos la función de evaluación sugerida anteriormente, porque no dudaría en sacrificar una reina ¡para capturar un peón!

Si la función de evaluación no es perfecta, una estrategia mejor es suponer que las negras van a responder con la jugada que minimice la función *eval*, puesto que cuanto más pequeño sea el valor de esta función, mejor será la jugada para las negras. Idealmente, las negras desearían un valor negativo muy grande. (Recordemos que llamamos jugada a cualquier acción, para evitar el término «media jugada».)

```
val ← −∞
para cada posición w que sea un sucesor de u hacer
    si w no tiene sucesor
      entonces  valw ← eval(w)
      sino valw ← mín{eval(x) | x es un sucesor de w}
    si valw ≥ val entonces val ← valw
                              v ← w
```

Ahora ya no aparece el problema consistente en sacrificar una reina para tomar un peón; naturalmente esta puede ser exactamente la regla equivocada si evita que las blancas encuentren la jugada ganadora. Quizá mirando más adelante el cambio pudiera resultar rentable. Por otra parte, seguro que evitamos aquellas jugadas que pudieran permitir a las negras dar mate inmediatamente (siempre y cuando *podamos* evitarlo).

Para añadir aspectos más dinámicos a la evaluación estática que proporciona *eval*, resulta preferible examinar varias jugadas por anticipado. Para examinar n jugadas por anticipado desde la posición u, las blancas deben pasar a la posición v dada por

$val \leftarrow -\infty$
para cada posición w que sea un sucesor de u **hacer**
    $B \leftarrow Negras(w, n)$
    si $B \leq val$ **entonces** $val \leftarrow B$
                $v \leftarrow w$

en donde las funciones *Blancas* y *Negras* son las siguientes

función *Negras*(w, n)
    si $n = 0$ **o** w no tiene sucesor
        entonces devolver $eval(w)$
        sino devolver $mín\{Blancas(x, n-1) \mid x$ es un sucesor de $w\}$

función *Blancas*(w, n)
    si $n = 0$ **o** w no tiene sucesor
        entonces devolver $eval(w)$
        sino devolver $máx\{Negras(x, n-1) \mid x$ es un sucesor de $w\}$

Ahora vemos la razón por la cual esta técnica se llama minimax. Las negras intentan minimizar la ventaja que se les permite a las blancas, y las blancas, por su parte, intentan maximizar la ventaja que se obtiene en cada jugada.

Con más generalidad, supongamos que la figura 9.18 muestra una parte del grafo correspondiente a algún juego. Si los valores asociados a los nodos del nivel más bajo se obtienen aplicando la función *eval* a las situaciones correspondientes, los valores para los otros nodos se pueden calcular empleando la regla de minimax. En el ejemplo, suponemos que el jugador A está intentando maximizar la función de evaluación, y que el jugador B está intentando minimizarla. Si A juega para maximizar su ventaja, seleccionará el segundo de los tres movimientos posibles. Esto le asegura un valor mínimo de 10; véase sin embargo el problema 9.55.

La técnica básica de minimax se puede mejorar en diferentes aspectos. Por ejemplo, quizá merezca la pena explorar con más profundidad los movimientos

más prometedores. De manera similar, la exploración de ciertas ramas se puede abandonar muy pronto si la información acerca de ellas de que se dispone es ya suficiente para mostrar que no pueden llegar a tener influencia en el valor de los nodos pertenecientes a zonas más altas del árbol. Este segundo tipo de mejora, que no describiremos en este libro, se conoce con el nombre de *poda alfa-beta*.

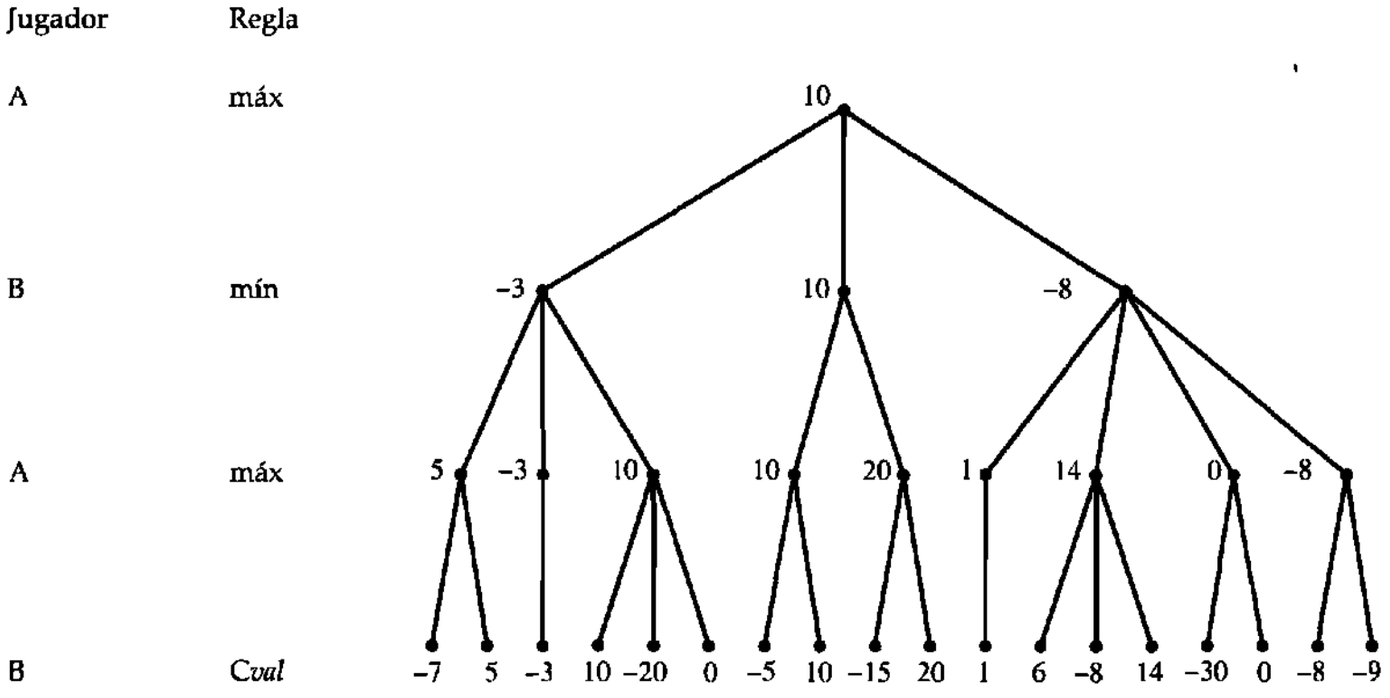


Figura 9.18. El principio de minimax

9.9 PROBLEMAS

Problema 9.1. Añadir los nodos $\langle 8, 7\rangle$, $\langle 7, 6\rangle$, $\langle 6, 5\rangle$ y sus descendientes al grafo de la figura 9.1.

Problema 9.2. ¿Es posible que una situación de victoria del juego descrito en la Sección 9.1 tenga más de una situación de derrota entre sus sucesores? En otras palabras, ¿existen situaciones en las que están disponibles varias jugadas ganadoras? ¿Puede suceder esto en el caso de una situación ganadora inicial $\langle n, n-1\rangle$?

Problema 9.3. Supongamos que se cambian las reglas del juego de la Sección 9.1 para que pierda el jugador que se vea obligado a tomar la última cerilla. Ésta es la versión *misère* del juego. Supongamos también que el primer jugador tiene que tomar al menos una cerilla, y que tiene que dejar dos como mínimo. Entre las situaciones iniciales con entre tres y ocho cerillas, ¿cuáles son ahora las situaciones ganadoras para el primer jugador?

Problema 9.4. Modificar el algoritmo *ganarec* de la Sección 9.1 para que devuelva un entero k, donde $k = 0$ si la situación es de derrota, y $1 \le k \le j$ si se produce una situación de victoria al tomar k cerillas.

Problema 9.5. Demostrar que en el juego descrito en la Sección 9.1, el primer jugador

dispone de una estrategia ganadora si y sólo si el número inicial de cerillas no aparece en la sucesión de Fibonacci.

Problema 9.6. Consideremos un juego que no pueda proseguir durante un número infinito de jugadas, y en que ninguna situación ofrezca al jugador al que corresponda el turno un número de jugadas válidas que sea infinito. Sea G el grafo dirigido que corresponde a este juego. Demostrar que el método descrito en la Sección 9.1 permite etiquetar todos los nodos de G como victoria, derrota o tablas.

Problema 9.7. Considere el juego siguiente. Inicialmente, se pone un montón de n cerillas en la mesa, entre dos jugadores. Cada jugador, por turnos, puede (a) partir cualquier montón de la mesa en dos montones desiguales o (b) quitar una o dos cerillas de cualquier montón de la mesa. No puede hacer las dos cosas a la vez. Sólo puede partir un montón, y si decide tomar dos cerillas, deben salir ambas del mismo montón. Gana el jugador que tome la última cerilla.

Por ejemplo, supongamos que durante el juego se llega a la situación {5, 4}, esto es, que hay dos montones en la mesa, uno con 5 cerillas y otro con 4. El jugador al que corresponda mover puede pasar a {4, 3, 2} o bien a {4, 4, 1} partiendo el montón de 5 cerillas; puede pasar a {5, 3, 1} partiendo el montón de 4 (pero no a {5, 2, 2} puesto que los nuevos montones deben de ser desiguales), o bien a {4, 4}, {4, 3}, {5, 3} o {5, 2} tomando una o dos cerillas de cualquiera de los montones.

Dibujar el grafo del juego para $n = 5$. Si ambos juegan correctamente, ¿gana el primer jugador o el segundo?

Problema 9.8. Repetir el problema anterior para la versión *misère* del juego, en que pierde el jugador que se quede con la última cerilla.

Problema 9.9. Consideremos un juego del tipo descrito en la Sección 9.1. Cuando utilizamos un grafo de posiciones de victoria y de derrota para describir este juego, presuponemos implícitamente que ambos jugadores moverán inteligentemente, para maximizar sus posibilidades de ganar. ¿Puede perder un jugador que esté en una situación de victoria si su oponente juega mal y comete un «error» inesperado?

Problema 9.10. Para cualquiera de las técnicas de recorrido de árboles que se mencionan en la Sección 9.2, demostrar que una implementación recursiva requiere un espacio de almacenamiento que está en $\Omega(n)$ en el caso peor.

Problema 9.11. Demostrar que cualquiera de las tres técnicas de recorrido de árboles mencionadas en la Sección 9.2 se puede implementar de tal forma que sólo necesite un tiempo que esté en $\Theta(n)$, y un espacio de almacenamiento en $\Theta(1)$, aun cuando los nodos no contengan un puntero a sus padres (en cuyo caso el problema es trivial).

Problema 9.12. Generalizar los conceptos de preorden y postorden a árboles arbitrarios (no binarios). Suponer que los árboles se representan como en la figura 5.7. Demostrar que estas dos técnicas siguen funcionando en un tiempo que está en el orden del número de nodos que haya en el árbol que hay que recorrer.

Problema 9.13. En la Sección 9.2 dábamos una forma de precondicionamiento para un árbol de tal forma que en lo sucesivo se pudiera verificar rápidamente si un nodo es o no un antecesor de otro. Existen varias maneras de llegar al mismo resultado. Demostrar, por ejemplo, que se puede hacer utilizando un recorrido en preorden seguido por un recorrido en preorden invertido, que visi-

ta primero un nodo y después sus subárboles de derecha a izquierda.

Si se representan los árboles tal como en la figura 5.7, ¿es este método más o menos eficiente que el dado en la Sección 9.2, o son comparables?

Problema 9.14. Mostrar cómo progresa un recorrido en profundidad a través del grafo de la figura 9.3 si el punto inicial es el nodo 6, y los vecinos de un nodo dado se visitan (a) en orden numérico y (b) en orden numérico decreciente.

Mostrar el árbol de recubrimiento y la numeración de los nodos del grafo generado por cada uno de estos recorridos.

Problema 9.15. Analizar el tiempo de ejecución del algoritmo *rp* si el grafo que hay que explorar se representa mediante una matriz de adyacencia (del tipo *grafoadya* de la Sección 5.4) en lugar de utilizar listas de nodos adyacentes.

Problema 9.16. Mostrar como se puede utilizar el recorrido en profundidad para hallar las componentes conexas de un grafo no dirigido.

Problema 9.17. Sea G un grafo no dirigido, y sea T el árbol de recubrimiento generado mediante un recorrido en profundidad de G. Demostrar que una arista de G que no tenga una arista correspondiente en T no puede unir nodos de distintas ramas del árbol, sino que necesariamente tiene que unir algún nodo v con uno de sus antecesores dentro de T.

Problema 9.18. Demostrar lo siguiente, o dar un contraejemplo:

(a) Si un grafo es biconexo, entonces es bicoherente.

(b) Si un grafo es bicoherente, entonces es biconexo.

Problema 9.19. Demostrar que un nodo v de un grafo conexo es un punto de articulación si y sólo si existen dos nodos a y b distintos de v tales que todo camino que una a con b pasa por v.

Problema 9.20. Demostrar que para cada par de nodos distintos u y v de un grafo biconexo existen al menos dos rutas que unen u con v y que no tienen nodos comunes salvo los nodos inicial y final.

Problema 9.21. En el algoritmo para hallar los puntos de articulación de un grafo no dirigido que se da en la Sección 9.3.1, mostrar cómo calcular los valores tanto de *prenum* como de *masalto* durante el recorrido en profundidad del grafo, e implementar el algoritmo correspondiente.

Problema 9.22. El ejemplo de la Sección 9.3.1 halla los puntos de articulación para el grafo de la figura 9.3 empleando un recorrido en profundidad que comienza en el nodo 1. Verificar que se encuentran los mismos puntos de articulación si el recorrido comienza en el nodo 6.

Problema 9.23. Ilustrar cómo funciona el algoritmo para hallar los puntos de articulación de un grafo no dirigido que se da en la Sección 9.3.1 sobre el grafo de la figura 9.19, empezando el recorrido (a) en el nodo 1 y (b) en el nodo 3.

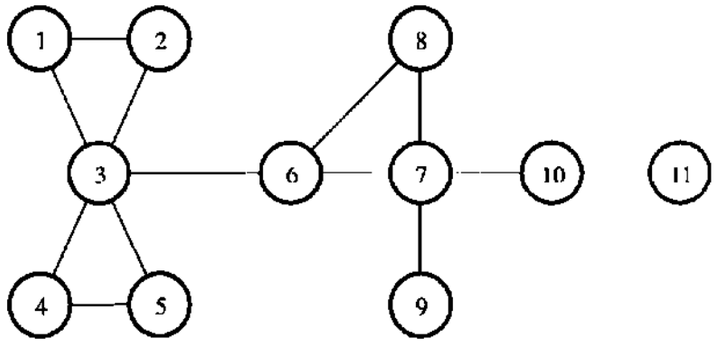


Figura 9.19. Un grafo con puntos de articulación

Problema 9.24. Un grafo es planar si se puede dibujar en un papel de tal manera que no se crucen ninguna de sus aristas. Utilizar un recorrido en profundidad para diseñar un algoritmo que sea capaz de decidir en tiempo lineal si un cierto grafo dado es o no planar.

Problema 9.25. Ilustrar el progreso del algoritmo de recorrido en profundidad aplicado al grafo de la figura 9.5 si el punto inicial es el nodo 1 y los vecinos de un nodo dado se examinan por orden numérico decreciente.

Problema 9.26. Sea F el bosque generado por un recorrido en profundidad en un grafo dirigido $G = \langle N, A \rangle$. Demostrar que G es acíclico si y sólo si el conjunto de aristas de G que no tienen una arista correspondiente en F no contiene ningún nodo que vaya de un nodo a alguno de sus antecesores dentro del bosque.

Problema 9.27. Para el grafo de la figura 9.8, ¿cuál es la ordenación topológica que se obtiene si se utiliza el procedimiento sugerido en la Sección 9.4.1, comenzando el recorrido en profundidad en el nodo 1 y visitando los vecinos de cada nodo por orden numérico?

Problema 9.28. ¿Qué falla en el argumento siguiente? Cuando se visita el nodo v de un grafo G empleando un recorrido en profundidad, se exploran inmediatamente todos los demás nodos que se puedan alcanzar desde v, siguiendo aristas de G. En la ordenación topológica, éstos otros nodos deben ir después de v. Por tanto, para obtener una ordenación topológica de estos nodos, basta con añadir una línea adicional

escribir v

al principio del procedimiento *rp*.

Problema 9.29. Un grafo dirigido es *fuertemente conexo* si existen caminos desde u hasta v y desde v hasta u para todo par de nodos u y v. Si un grafo dirigido no es fuertemente conexo, los mayores conjuntos de nodos tales que los subgrafos inducidos son fuertemente conexos se denominan *componentes fuertemente conexos* del grafo. Por ejemplo, los componentes fuertemente conexos del grafo de la figura 9.5 son {1, 2, 3}, {5, 6} y {4, 7,8}. Diseñar un algoritmo eficiente, basado en un recorrido en profundidad, para hallar las componentes fuertemente conexas de un grafo.

Problema 9.30. Después de un recorrido en anchura de un grafo no dirigido, sea F el bosque de árboles generado. Si $\{u, v\}$ es un arista de G que no posee una arista correspondiente en F (estas aristas se representan mediante líneas discontinuas en la figura 9.10), mostrar que los nodos u y v yacen en el mismo árbol de F, pero que ni u ni v son antecesores uno de otro.

Problema 9.31. Mostrar cómo progresa un recorrido en anchura a través del grafo de la figura 9.5, suponiendo que los vecinos de cada nodo se visiten en orden numérico, y que los puntos iniciales necesarios se seleccionen también por orden numérico.

Problema 9.32. Dibujar el bosque generado por el recorrido del problema 9.31, mostrando las aristas restantes del grafo mediante líneas discontinuas. ¿Cuántas clases de líneas «discontinuas» puede haber?

Problema 9.33. Justificar la afirmación de que un recorrido en profundidad del grafo de la figura 9.11, comenzando en el nodo 1 y visitando los vecinos por el orden «primero dividir por 3, luego multiplicar por 2», baja hasta el nodo 12 y después visita sucesivamente los nodos 24, 48, 96, 32 y 64.

Problema 9.34. Enumerar los quince primeros nodos visitados por un recorrido en anchura del grafo de la figura 9.11, comenzando en el nodo 1 y visitando los vecinos por el orden «primero dividir por 3, luego multiplicar por 2».

Problema 9.35. La Sección 9.5 da una forma de producir el valor 13, empezando desde 1 y utilizando las operaciones de multiplicación por 2 y división por 3. Encontrar otra forma de producir el valor 13 utilizando el mismo punto inicial y las mismas operaciones.

Problema 9.36. Un nodo p de un grafo dirigido $G = \langle N, A \rangle$ se denomina sumidero si para todo nodo $v \in N$, $v \neq p$, existe la arista (v, p), mientras que no existe la arista (p, v). Escribir un algoritmo que pueda detectar la presencia de un sumidero en G en un tiempo que esté en $O(n)$, en donde n es el número de nodos que haya en el grafo. El algoritmo debe aceptar el grafo representado por su matriz de adyacencia (del tipo *grafoadya* de la Sección 5.4). Obsérvese que un tiempo de ejecución en $O(n)$ para este problema es algo notable, dado que sólo para escribir los datos se requiere un espacio que está en $\Omega(n^2)$.

Problema 9.37. Un camino de Euler en un grafo no dirigido es un camino en la cual todas las aristas aparecen una sola vez. Diseñar un algoritmo que determine si un grafo dado posee un camino de Euler, y que imprima ese camino si existe. ¿Cuánto tiempo necesita su algoritmo?

Problema 9.38. Repetir el problema 9.37 para un grafo dirigido.

Problema 9.39. Dado un grafo dirigido o no dirigido, se dice que un camino es hamiltoniano cuando pasa exactamente una vez por todos los nodos del grafo, sin volver al nodo inicial. En un grafo dirigido, sin embargo, es preciso tener en cuenta la dirección de las aristas en el camino. Demostrar que si un grafo dirigido es completo (esto es, si todo par de nodos está unido al menos en una dirección) entonces tiene un camino hamiltoniano. En este caso, dar un algoritmo para encontrar el camino.

Problema 9.40. Dibujar el árbol de búsqueda explorado por un algoritmo de vuelta atrás que resuelva el mismo caso del problema de la mochila de la Sección 9.6.1, pero suponiendo esta vez que los elementos se cargan en orden decreciente de pesos.

Problema 9.41. Resolver el mismo caso del problema de la mochila de la Sección 9.6.1 por programación dinámica. Habrá que hacer primero el problema 8.5.1.

Problema 9.42. Adaptar la función *mochila* de la Sección 9.6.1 para dar la composición de la carga óptima, además de su valor.

Problema 9.43. Supongamos que el vector $q[1..8]$ representa las posiciones de ocho reinas en un tablero de ajedrez, con una reina en cada fila: si $q[i] = j$, $1 \leq i \leq 8$, $1 \leq j \leq 8$, la reina de la fila i está en la columna j. Escribir una función *solución(q)* que devuelva el valor falso si al menos una de las reinas amenaza a otra, y que devuelva el valor verdadero en caso contrario.

Problema 9.44. Dado que el algoritmo *reinas1* halla una solución y se detiene después de comprobar 1.299.852 situaciones, resolver el problema sin emplear una computadora.

Problema 9.45. Supongamos que el procedimiento *usar(T)* que se invoca desde el procedimiento *perm(i)* de la Sección 9.6 consiste simplemente en imprimir la matriz T en una línea nueva. Mostrar el resultado de llamar a *perm(1)* cuando $n = 4$.

Problema 9.46. Supongamos que el procedimiento *usar(T)* que se invoca desde el procedimiento *perm(i)* de la Sección 9.6 requiere un tiempo constante. ¿Cuánto tiempo se necesita, en función de n, para ejecutar la llamada *perm(1)*?

Rehacer el problema suponiendo ahora que *usar*(*T*) requiere un tiempo que está en $\Theta(n)$.

Problema 9.47. ¿Para qué valores de n carece de solución el problema de las n reinas? Demostrar la respuesta.

Problema 9.48. ¿Cuántas soluciones tiene el problema de las ocho reinas? ¿Cuántas soluciones *distintas* hay si no distinguimos las soluciones que se pueden transformar unas en otras por rotaciones y reflexiones?

Problema 9.49. Investigar el problema consistente en situar k «reinas» en un tablero tridimensional $n \times n \times n$. Suponga que una «reina» en tres dimensiones amenaza las posiciones de las mismas filas, columnas o diagonales en que se encuentra, de la forma evidente. Claramente, k no puede sobrepasar el valor n^2. Sin contar el caso trivial $n = 1$, ¿cuál es el valor más pequeño de n para el cual es posible una solución con n^2 «reinas»?

Problema 9.50. Una matriz booleana $M[1..n, 1..n]$ representa un laberinto cuadrado. En general, partiendo de un punto dado, podemos pasar a puntos adyacentes de la misma fila o de la misma columna. Si $M[i, j]$ es *verdadero*, se puede pasar por el punto (i, j); si $M[i, j]$ es *falso*, no se puede pasar por el punto (i, j). La figura 9.20 da un ejemplo.

Dar un algoritmo de vuelta atrás que busque un camino, si existe, desde (1,1) hasta (n, n). Sin ser completamente formal,por ejemplo, se pueden decir cosas como «**para** todo punto v que sea un vecino de x ***hacer***...», el algoritmo debe ser claro y preciso.

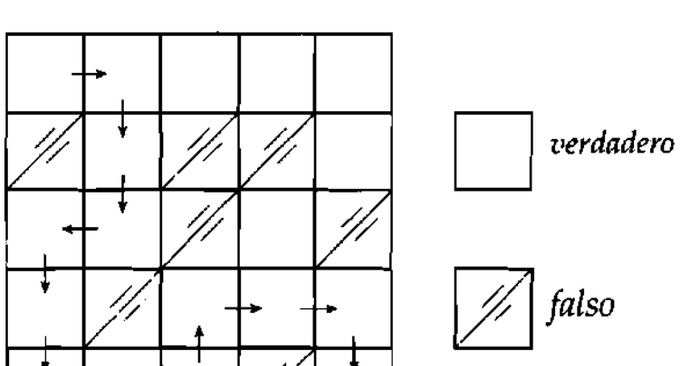


Figura 9.20. Un laberinto

Problema 9.51. El método de vuelta atrás sugerido en el problema 9.50 ilustra el principio, pero resulta muy ineficiente en la práctica. Buscar una forma mucho mejor de resolver el mismo problema.

Problema 9.52. En la Sección 9.7 se calculaban las cotas inferiores de los nodos del árbol de búsqueda suponiendo que toda tarea no asignada sería ejecutada por el agente sin asignación que pudiera hacerlo con menor coste. Esto es lo mismo que tachar las filas y columnas de la matriz de costes que correspondan a los agentes y tareas que ya tengan asignación, y sumar entonces los elementos mínimos de cada una de las columnas restantes. Una forma alternativa de calcular las cotas es suponer que todo agente sin asignación efectuará la tarea que pueda realizar con el mínimo coste. Esto es como tachar las filas y columnas de la matriz de costes correspondientes a los agentes y tareas que ya tengan asignación, y sumar después los elementos mínimos de todas las filas restantes. Mostrar cómo funciona un algoritmo de ramificación y poda para el caso de la figura 9.7 actúa utilizando este segundo método de calcular cotas inferiores.

Problema 9.53. Utilizar un algoritmo de ramificación y poda para resolver los problemas de asignación que tienen asociadas las siguientes matrices:

(a)

	1	2	3	4
a	94	1	54	68
b	74	10	88	82
c	62	88	8	76
d	11	74	81	21

(b)

	1	2	3	4	5
a	11	17	8	16	20
b	9	7	12	6	15
c	13	16	15	12	16
d	21	10	12	11	15
e	14	10	12	11	15

Problema 9.54. Se dispone de cuatro tipos de objetos, cuyos pesos son respectivamente 7, 4, 3 y 2 unidades, y cuyos valores son 9, 5, 3 y 1. Podemos transportar un máximo de 10 unidades de peso. Los objetos no se pueden fragmentar en trozos. Determinar la carga más valiosa que se pueda transportar, empleando (a) programación dinámica y (b) ramificación y poda. Para la parte (a), es necesario resolver primero el problema 8.51.

Problema 9.55. Examinando el árbol de la figura 9.18, dijimos que el jugador que va a mover esta situación tiene asegurado un valor de 10 como mínimo. ¿Es esto estrictamente cierto?

Problema 9.56. Supongamos que u corresponde a la situación inicial de las piezas en el juego del ajedrez. ¿Qué se puede decir acerca de *blancas*(u,12345), además del hecho de que tardaríamos demasiado tiempo en calcularlo, incluso empleando una computadora especializada? Justifique su respuesta.

Problema 9.57. El juego de tres en raya tridimensional se juega en un «tablero» 4 ×4 ×4. Tal como en el tres en raya convencional, los jugadores *X* y *O* van marcando cuadrados alternativamente. El ganador es el primero que alinea cuatro de sus propias marcas en cualquier dirección, bien sea paralela a un lado del tablero o en una diagonal. Hay 76 maneras de ganar. Diseñar una función de evaluación para situaciones de este juego, y utilizarla para escribir un programa que juegue contra un oponente humano.

9.10 REFERENCIAS Y TEXTOS MÁS AVANZADOS

Existe un buen número de libros acerca de algoritmos para grafos o problemas de combinatoria que suelen plantearse en términos de grafos. Mencionaremos por orden cronológico a *Christofides* (1975), *Lawler* (1976), *Reingold, Nievergelt* y *Deo* (1977), *Gondran* y *Minoux* (1979), *Even* (1989), *Papadimitriou* y *Steiglitz* (1982), *Tarjan* (1983) y *Melhorn* (1984b). La noción matemática de grafo se trata extensamente en *Berge* (1958, 1970).

Berlekamp, Conway y *Guy* (1982) dan más versiones del juego de *Nim* que las que interesarán a la mayoría. El juego del problema 9.7 es una variante del juego de *Grundy*, que también se discute en *Berlekamp, Conway* y *Guy* (1982). El libro de *Nilsson* (1971) es una fuente inagotable de

ideas acerca de grafos y juegos, de la técnica de minimax y de la poda alfa-beta. En Good (1968) aparecen algunos algoritmos para jugar al ajedrez. Una grata descripción de la primera vez que un programa de computadora derrotó a un campeón mundial de *backgammon* se da en Deyong (1977). Para una descripción más técnica de esta hazaña, consúltese Berliner (1980). En 1994, Gari Kasparov, el campeón del mundo de ajedrez, fue derrotado por una microcomputadora Pentium. En el momento de escribir estas líneas, los seres humanos siguen invictos en el juego.

Una solución del problema 9.11 se da en Robson (1973). Para aprender más acerca del precondicionamiento, véase Brassard y Bratley (1988). Se pueden hallar muchos algoritmos basados en el recorrido en profundidad en Tarjan (1972), Hopcroft y Tarjan (1973), y Aho, Hopcroft y Ullman (1974, 1983). El problema 9.24 está resuelto en Hopcroft y Tarjan (1974). Véase también Rosenthal y Goldner (1977) para un algoritmo eficiente que, dado un grafo no dirigido que sea conexo pero no biconexo, busca un conjunto mínimo de aristas que se pueden añadir para hacer que el grafo sea biconexo.

El problema que implica multiplicar por 3 y dividir por 2 nos recuerda el problema de Collatz: véase el problema E16 en Guy (1981). La vuelta atrás se describe en Golomb y Baumert (1965), y se dan técnicas para analizar su eficiencia en Knuth (1975). El problema de las ocho reinas fue inventado por Bezzel en 1848; véase la descripción de Ball (1967). Irving (1984) da un algoritmo de vuelta atrás especialmente eficiente para hallar todas las soluciones del problema de las ocho reinas. El problema se puede resolver en tiempo lineal mediante un algoritmo de divide y vencerás debido a Abramson y Yung (1989), siempre que nos conformemos con una única solución. Volveremos a este problema en el Capítulo 10. El problema tridimensional de n^2 reinas mencionado en el problema 9.49 fue mencionado por McCarty (1978) y resuelto por Allison, Yee y McGaughey (1989): no hay soluciones para $n < 11$ pero existen 264 soluciones para $n = 11$.

La ramificación y poda se explica en Lawler y Wood (1966). El problema de la asignación es muy conocido en investigación operativa: véase Hillier y Lieberman (1967). Para un ejemplo de resolución del problema de la mochila empleando vuelta atrás, véase Hu (1981). El problema del viajante de comercio se resuelve empleando ramificación y poda en Bellmore y Nemhauser (1968).

Capítulo 10

Algoritmos probabilistas[1]

10.1 INTRODUCCIÓN

Imagine que es usted la heroína de un cuento de hadas. Hay un tesoro oculto en un lugar que se describe en un mapa que no consigue descifrar. Se las ha arreglado para reducir la búsqueda a dos posibles escondites, que sin embargo distan bastante entre sí. Si estuviera en uno u otro de estos dos lugares, sabría de inmediato si es o no el correcto. Se necesitan cinco días para ir a cualquiera de los dos lugares, o para viajar desde uno hasta otro. Y el problema se complica más, porque hay un dragón que visita el tesoro todas las noches, y se lleva una parte a un inaccesible cubil en las montañas. Piensa usted que se necesitarán cuatro días de cálculo para resolver el misterio del mapa, y por tanto para saber con certeza dónde está oculto el tesoro, pero si se pone en marcha ya no podrá utilizar una computadora. Un elfo le ofrece mostrarle la forma de descifrar el mapa a cambio del equivalente del tesoro que se lleva el dragón en tres noches. ¿Debe de aceptar la oferta del elfo?

Evidentemente, es preferible dar el valor de tres noches de tesoro al elfo frente a permitir que el dragón se dedique al pillaje cuatro noches más. Sin embargo, si está dispuesta a correr un riesgo calculado, puede hacerlo mejor. Supongamos que es x el valor del tesoro que queda hoy, y que y es el valor del tesoro que se lleva el dragón todas las noches. Supongamos además que es $x > 9y$. Recordando que necesitará cinco días para alcanzar el escondite, puede esperar volver a casa con $x–9y$ si espera cuatro días para acabar de descifrar el mapa. Si acepta la oferta del elfo, puede partir inmediatamente y volver con $x – 5y$, de lo cual $3y$ servirá para pagar al elfo; entonces le quedará $x–8y$. Una estrategia mejor consiste en tirar una moneda al aire para decidir cuál de los dos escondites va a visitar primero, viajando hasta el otro si se ha equivocado. Esto le da una oportunidad de entre dos de volver a casa con $x – 5y$, y una oportu-

nidad de entre dos de volver a casa con $x - 10y$. Por tanto, el beneficio esperado es $x - 7{,}5y$.

Esta fábula se puede traducir al contexto de la algoritmia en la forma siguiente: cuando un algoritmo se enfrenta a una decisión, a veces es preferible seguir un curso de acción aleatorio, en lugar de invertir tiempo en determinar cuál de las alternativas es la mejor. Esta situación se produce cuando el tiempo necesario para determinar la opción óptima es prohibitivo en comparación con el tiempo que se ahorra de análisis en el caso medio al tomar esta decisión óptima.

La característica fundamental de los algoritmos probabilistas es que un mismo algoritmo se puede comportar de forma distinta cuando se aplica dos veces a un mismo caso. Su tiempo de ejecución, e incluso el resultado obtenido, pueden variar considerablemente entre usos consecutivos. Esto se puede explotar de muchas maneras. Por ejemplo, no se permite que un algoritmo determinista pierda el control (bucle infinito, división por cero) porque si hace esto en un caso dado, entonces nunca se podrá resolver ese caso con ese algoritmo. Por contraste, este comportamiento es admisible para un algoritmo probabilista siempre y cuando se produzca con una probabilidad razonablemente pequeña en cualquier caso dado: si el algoritmo se atasca, basta reiniciarlo para ese mismo caso y dispondremos de una nueva oportunidad de éxito. Otra ventaja de este enfoque es que si hay más de una respuesta correcta, se pueden obtener varias diferentes ejecutando el algoritmo probabilista más de una vez; un algoritmo determinista siempre llega a la misma respuesta, aunque por supuesto se puede programar para buscar varias.

Otra consecuencia del hecho de que los algoritmos probabilistas se pueden comportar de forma distinta cuando se ejecutan dos veces con las mismas entradas es que algunas veces les permitiremos que produzcan resultados erróneos. Supuesto que esto suceda con probabilidad suficientemente pequeña en cualquier caso dado, basta invocar el algoritmo varias veces sobre el caso deseado para llegar a una confianza arbitrariamente grande de que se ha obtenido la respuesta correcta. Por contraste, un algoritmo determinista que da respuestas erróneas con algunas entradas es inadmisible para la mayoría de las aplicaciones, porque *siempre* fallará en esas entradas.

El análisis de algoritmos probabilistas suele ser complicado, y requiere estar familiarizado con resultados de probabilidad y estadística que van más allá de los presentados en la Sección 1.7.4. Por esta razón, en este capítulo se citan varios resultados sin demostración.

10.2 PROBABILISTA NO IMPLICA INCIERTO

Hay muchos algoritmos probabilistas que dan respuestas probabilistas, esto es, respuestas que no son necesariamente exactas. Hay aplicaciones críticas para las cuales no se pueden admitir esas respuestas inciertas. Sin embargo, es frecuente que la probabilidad de error se pueda hacer descender por debajo de la de un

error del equipo físico durante el significativamente mayor tiempo que el necesario para calcular la respuesta de forma determinista. Por tanto, paradójicamente, ¡puede suceder que la respuesta incierta dada por un algoritmo probabilista no sólo se obtenga más deprisa, sino que además podamos confiar más en ella que en la respuesta «garantizada» que se obtiene mediante ningún algoritmo determinista! Además, hay problemas para los cuales no se conoce ningún algoritmo (determinista o probabilista) que pueda dar la respuesta con certeza en una cantidad razonable de tiempo (incluso ignorando posibles errores del equipo físico), y sin embargo hay algoritmos probabilistas que pueden resolver rápidamente el problema si se admite una pequeña probabilidad de error. Tal es el caso, por ejemplo, si se desea determinar si un número de 1.000 dígitos es primo o compuesto.

Hay dos categorías de algoritmos probabilistas que no garantizan la corrección del resultado. Los algoritmos *numéricos* producen un *intervalo de confianza* de la forma «con una probabilidad del 90%, la respuesta correcta es 59 más o menos 3». Cuanto más tiempo se conceda a estos algoritmos numéricos, más preciso es el intervalo. Las encuestas de opinión ofrecen un ejemplo familiar de este tipo de respuesta. Además ilustran que un algoritmo probabilista puede ser mucho más eficiente, aunque menos preciso, que el correspondiente algoritmo determinista (una elección general). Los algoritmos numéricos son útiles si nos basta una aproximación de la respuesta correcta. En contraste, los denominados algoritmos de *Monte Carlo* dan la respuesta exacta con una elevada probabilidad, aunque a veces proporcionan una respuesta incorrecta. En general, no se puede saber si la respuesta proporcionada por un algoritmo de Monte Carlo es correcta, pero se puede reducir arbitrariamente la probabilidad de error dando más tiempo al algoritmo. (Por razones históricas, algunos autores utilizan el término «Monte Carlo» para cualquier algoritmo probabilista, y en particular para los que nosotros denominamos numéricos.)

Sin embargo, no todos los algoritmos probabilistas dan respuestas probabilísticas. Con frecuencia las opciones probabilistas se utilizan solamente como guía para llevar más rápidamente el algoritmo hacia la solución deseada, en cuyo caso la respuesta obtenida siempre es correcta. Además, puede ocurrir que la corrección de toda presunta solución pueda ser verificada eficientemente. Si resulta ser incorrecta, el algoritmo probabilista debería admitir que ha fracasado, en lugar de responder incorrectamente. Los algoritmos probabilistas que nunca dan una respuesta incorrecta se denominan de *Las Vegas*. Una vez más la posibilidad de que el algoritmo pueda admitir un fallo no es dramática: basta seguir aplicando el algoritmo a la entrada deseada hasta que tenga éxito.

Quizá resulte humorístico, pero ilustraremos la diferencia entre los distintos tipos de algoritmos probabilistas por la forma en que responderían a la pregunta ¿cuándo descubrió América Cristóbal Colón? Si lo invocásemos cinco veces, un algoritmo numérico podría responder:

Entre 1490 y 1500
Entre 1485 y 1495
Entre 1491 y 1501
Entre 1480 y 1490
Entre 1489 y 1499

Si le damos más tiempo al algoritmo, podemos reducir la probabilidad de un intervalo falso (parece ser un 20% en nuestra muestra, puesto que la cuarta respuesta no era correcta) o bien podemos reducir la amplitud de los intervalos (11 en nuestro ejemplo), o las dos cosas. Cuando se le hace la misma pregunta diez veces a un algoritmo de Monte Carlo, podría responder

1492, 1492, 1492, 1491, 1492, 1492, 357 a.C., 1492, 1492, 1492

Una vez más, tenemos una tasa de error del 20% en la respuesta, que se podría hacer más pequeña dando más tiempo al algoritmo. Obsérvese que en algunas ocasiones las respuestas incorrectas están cerca de la respuesta correcta, y a veces son completamente erróneas. Por último, un algoritmo de Las Vegas invocado 10 veces podría responder

1492, 1492, !Perdón!, 1492, 1492, 1492, 1492, 1492, ¡Perdón!, 1492

El algoritmo nunca da una respuesta incorrecta, pero de vez en cuando no responde.

10.3 TIEMPO ESPERADO FRENTE A TIEMPO PROMEDIO

Haremos una distinción importante entre las palabras «promedio» y «esperado». El tiempo *promedio* de un algoritmo determinista se describió en la Sección 2.4. Se refiere al tiempo promedio requerido por el algoritmo cuando se consideran igualmente probables todos los ejemplares posibles de un tamaño dado. Tal como se vio, el análisis del caso promedio puede inducir a confusión si la realidad es que algunos ejemplares son más probables que otros. Por contraste, el tiempo *esperado* de un algoritmo probabilista se define para cada ejemplar individual: se refiere al tiempo promedio que se necesitaría para resolver ese mismo ejemplar una y otra vez. Ésta es una noción robusta, que no depende de la distribución de probabilidades de los ejemplares que haya que resolver, porque las posibilidades implicadas quedan bajo control directo del algoritmo.

Resulta significativo hablar del tiempo esperado promedio y del tiempo esperado en el caso peor de un algoritmo probabilista. Este último, por ejemplo, se refiere al tiempo esperado que requiere el peor caso posible de un tamaño dado, y *no* al tiempo requerido si lamentablemente se tomasen las peores opciones probabilistas. Los algoritmos de Las Vegas pueden ser más eficientes que los algoritmos deterministas, pero sólo con respecto al tiempo esperado: siempre es posible que la mala fortuna obligue a un algoritmo de Las Vegas a seguir el camino más largo para llegar a una solución. Sin embargo, el comportamiento esperado de los algoritmos de Las Vegas puede ser mucho mejor que el de los algoritmos deterministas.

Por ejemplo, en la Sección 7.5 vimos un algoritmo que puede hallar la mediana de un vector en un tiempo lineal en el caso peor. En la Sección 10.7.2 estudiaremos un algoritmo de Las Vegas para hallar la mediana cuyo tiempo esperado en el caso peor también es lineal con respecto al tamaño del caso. Su rendimiento esperado es considerablemente mejor que el del algoritmo de la Sección 7.5 *en todos y cada uno de los casos*. El único precio que hay que pagar cuando se utiliza este algoritmo de Las Vegas es la improbable posibilidad de que en algunas ocasiones requiera un tiempo excesivo, independientemente del caso en cuestión, y esto no se deberá a otra cosa que a la mala suerte.

Se puede emplear un enfoque parecido para transformar *quicksort* en un algoritmo que ordena n elementos en un tiempo esperado en el caso peor que se encuentra en $O(n \log n)$, mientras que el algoritmo que vímos en la Sección 7.4.2 requiere un tiempo que está en $\Omega(n^2)$ cuando los elementos ya están ordenados. De forma similar, la *dispersión universal* ofrece acceso a una tabla de símbolos en un tiempo esperado constante en el caso peor, mientras que la dispersión clásica resulta desastrosa en el caso peor; véase la Sección 10.7.3. La ventaja principal de hacer aleatorio un algoritmo determinista que ofrezca un buen comportamiento medio a pesar de un caso peor muy malo es hacerlo menos vulnerable a una distribución de probabilidad inesperada de los casos que intente resolver alguna aplicación; véase el final de la Sección 2.4 y la Sección 10.7.

10.4 GENERACIÓN DE NÚMEROS PSEUDOALEATORIOS

A lo largo de este capítulo, suponemos que está disponible un generador de números aleatorios que se puede invocar con coste unitario. Sean a y b dos números reales tales que $a < b$. Una llamada a *uniforme*(a, b) devuelve un número real x seleccionado aleatoriamente en el intervalo $a \leq x < b$. La distribución de x es uniforme en el intervalo, y las sucesivas llamadas al generador devuelve valores independientes de x. Para generar enteros aleatorios, extendemos la notación a *uniforme*$(i..j)$, en donde i y j son enteros, $i \leq j$, y la llamada devuelve un entero k seleccionado aleatoriamente, uniformemente e independientemente dentro del intervalo $i \leq k \leq j$. Un caso especial útil es la función *tiramoneda*, que devuelven *cara* o *cruz*, cada una de ellas con una probabilidad del 50%. Obsérvese que *uniforme*(a, b), *uniforme*$(i..j)$ y *tiramoneda* son fáciles de obtener aun cuando sólo esté disponible *uniforme*$(0, 1)$:

```
función uniforme(a,b)
    devolver a + (b − a) × uniforme(0, 1)

función uniforme(i..j)
    devolver ⌊uniforme(i, j + 1)⌋

función tiramoneda
    si uniforme(0..1) = 0 entonces devolver cara
                          sino devolver cruz
```

Los generadores verdaderamente aleatorios no suelen estar disponibles en la práctica. Además, no es realista esperar que *uniforme*(0, 1) tome un valor *real* arbitrario entre 0 y 1. En la mayoría de las ocasiones, se emplean generadores *pseudoaleatorios*: se trata de procedimientos deterministas capaces de generar largas sucesiones de valores que parecen tener las propiedades de una sucesión aleatoria. Para dar comienzo a una sucesión, proporcionamos un valor inicial denominado *semilla*. Una misma semilla da lugar siempre a la misma sucesión. Para obtener sucesiones diferentes, podemos seleccionar, por ejemplo, una semilla que dependa de la fecha o de la hora. Casi todos los lenguajes de programación contienen uno de estos generadores, aunque hay que utilizar con precaución algunas implementaciones. Empleando un buen generador pseudoaleatorio, podemos esperar que sean válidos los resultados teóricos obtenidos en este capítulo concernientes a la eficiencia y fiabilidad de diferentes algoritmos probabilistas. Sin embargo, la poco práctica hipótesis de que está disponible un generador verdaderamente aleatorio resulta crucial cuando se efectúa el análisis.

La teoría de los generadores pseudoaleatorios va más allá de los límites de este libro, pero el principio general es sencillo. La mayoría de los generadores está basada en una pareja de funciones $f : X \to X$ y $g : X \to Y$, donde X es un conjunto suficientemente grande e Y es el dominio de los valores pseudoaleatorios que haya que generar. Se impone que tanto X como Y sean finitos, lo cual significa que sólo podemos tener la esperanza de aproximar el efecto de *uniforme*(0, 1) empleando un Y adecuadamente grande. Por otra parte, $Y = \{0, 1\}$ resulta adecuado para muchas aplicaciones: corresponde a tirar una moneda imparcial. Para generar nuestra secuencia pseudoaleatoria, sea $s \in X$ una semilla. Esta semilla define una secuencia:

$$\begin{cases} x_0 = s \\ x_i = f(x_{i-1}) \text{ para todo } i > 0 \end{cases}$$

Finalmente, la sucesión pseudoaleatoria $y_0, y_1, y_2, \ldots$ se define de la forma $y_i = g(x_i)$ para todo $i \geq 0$. Esta sucesión es esencialmente periódica, con un periodo que no puede sobrepasar el número de elementos que hay en X. Sin embargo, si X es suficientemente grande y si f y g (y a veces s) se seleccionan adecuadamente, entonces el periodo puede hacerse muy grande, y la sucesión puede ser, para casi todos los efectos prácticos, indistinguible de una sucesión verdaderamente aleatoria de elementos de Y.

Daremos un ejemplo sencillo para ilustrar este principio. Sean p y q dos números primos muy grandes distintos, ambos congruentes con 3 módulo 4, y sea n su producto. Supongamos que p y q son tan grandes que no resulta factible factorizar n. (En el momento de escribir esto, se consideran suficientes 100 cifras para cada número; véase la Sección 10.7.4.) Sea X el conjunto de enteros desde 0 hasta n–1, y sea $Y = \{0, 1\}$. Definimos $f(x) = x^2 \bmod n$ y $g(x) = x \bmod 2$. Siempre y cuando la semilla se seleccione uniformemente al azar entre los elementos de X que son relativamente primos con respecto a n, se ha demostrado que casi siempre es imposible distinguir la sucesión generada de esta manera de una sucesión binaria verdaderamente aleatoria. Para la mayoría de los propósitos algorítmicos, pueden resultar preferibles unos generadores pseudoaleatorios más rápidos pero menos seguros. Por ejemplo,

gozan de amplia aceptación los generadores pseudoaleatorios de congruencia lineal de la forma $f(x) = ax + b \bmod n$, para valores adecuados de a, b y n.

10.5 ALGORITMOS PROBABILISTAS NUMÉRICOS

La aleatoriedad se utilizó por primera vez en algoritmia para resolver aproximadamente problemas numéricos. Ya se utilizaba en el mundo secreto de la investigación atómica durante la Segunda Guerra Mundial, momento en que se dio a esta técnica el nombre de «Monte Carlo» (término que algunos autores utilizan todavía para denotar los algoritmos probabilistas numéricos, pero que emplearemos con un significado diferente en este libro). Un ejemplo familiar de un algoritmo numérico probabilista es la simulación. Esta técnica se puede emplear, por ejemplo, para estimar la longitud media de la cola en un sistema tan complejo que no sea factible obtener soluciones en forma cerrada, o incluso respuestas numéricas por métodos deterministas. Para ciertos problemas de la vida real, el cálculo de una solución exacta no es posible ni siquiera en principio, quizá debido a incertidumbres en los datos experimentales que haya que utilizar, o quizá porque una computadora digital sólo puede manejar valores binarios o decimales mientras que la respuesta que hay que calcular es irracional. Para otros problemas, existe una respuesta precisa pero se tardaría demasiado tiempo en calcularla exactamente.

La respuesta obtenida mediante un algoritmo probabilista numérico siempre es aproximada, pero su precisión esperada mejora a medida que crece el tiempo disponible para el algoritmo. El error suele ser inversamente proporcional a la raíz cuadrada de la cantidad de trabajo que hayamos efectuado, así que se necesita 100 veces más trabajo para obtener un dígito de precisión adicional.

10.5.1 La aguja de Buffon

En el siglo XVIII, Georges Louis Leclerc, conde de Buffon, demostró que si se arroja aleatoriamente una aguja (en una posición aleatoria, con un ángulo aleatorio, con una distribución uniforme) sobre un suelo hecho con planchas de anchura constante, si la aguja tiene una longitud que sea exactamente la mitad de la anchura de las planchas, y si la anchura de la rendija existente entre las planchas es cero, entonces la probabilidad de que una aguja caiga encima de una rendija es $1/\pi$. Este teorema puede servir para predecir aproximadamente cuántas agujas caerán encima de una rendija si dejamos caer n de ellas: la esperanza es n/π. La figura 10.1 ilustra la caída de 22 agujas, de las cuales 8 han caído encima de una rendija; nosotros habríamos esperado 7 análisis en el caso medio, puesto que es $\pi \approx {}^{22}/_{7}$. Suponemos que las agujas se dejan caer al azar e independientemente unas de otras: para evitar interferencias, no deben superponerse, ni siquiera deben tocarse. Quizá sea mejor tirar y recoger la misma aguja varias veces.

Sin embargo, existe una interpretación del teorema de Buffon que resulta algorítmicamente más interesante. En lugar de utilizarlo para predecir el número de agujas que caerán encima de una rendija (reconocemos que esto no tiene demasiado interés), ¿por qué no utilizarlo al revés, para estimar el valor de π? Suponga

que no dispone de una buena aproximación de π, pero conoce el teorema de Buffon. Imagine que deja caer un gran número de agujas, n, en el suelo, y que cuenta el número de agujas k que caen encima de una rendija. Ahora puede utilizar n/k como estimación de π. (Sería tener mala suerte si $k = 0$, pero esto resulta muy poco probable cuando n es grande.) Cuanto más precisa quiera que sea su estimación, más agujas tendrá que dejar caer. En principio, puede aproximarse cuanto quiera al valor de π sin más que dejar caer un número suficiente de agujas, o preferiblemente dejar caer una misma aguja un número suficiente de veces para evitar interferencias. Llevaremos a cabo un análisis más preciso más adelante en esta sección. En la práctica, esto no es un «algoritmo» útil, porque se pueden obtener aproximaciones de π mucho mejores empleando métodos deterministas. Sin embargo, esta aproximación a la «determinación experimental de π» fue muy utilizada en el siglo XIX, haciendo de éste uno de los primeros algoritmos probabilistas que se utilizaron.

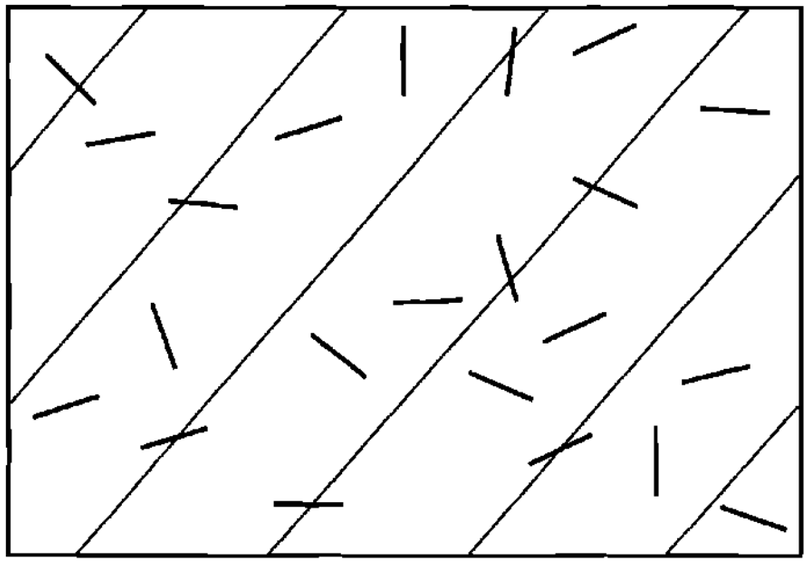


Figura 10.1 La aguja de Buffon

Hemos visto que el teorema de Buffon puede servir para predecir el número de agujas que caen encima de una rendija, y para estimar el valor de π. Una tercera aplicación posible para este teorema es estimar la anchura de las planchas (en términos de la longitud de la aguja). Recordemos que las dos aplicaciones anteriores daban por supuesto que las agujas son *precisamente* de longitud igual a la mitad de la anchura de las planchas. Si no es cierta esta suposición, entonces ya no es cierto que n/k se aproxime arbitrariamente a π a medida que vayamos tirando más y más agujas. Sean w y λ la anchura de las planchas y la longitud de las agujas, respectivamente. Buffon demostró un teorema más general: supuesto que sea $w \geq \lambda$, la probabilidad de que una aguja tirada aleatoriamente caiga encima de una rendija es $p = 2\lambda/w\pi$. En particular, $p = 1/\pi$ cuando $\lambda = w/2$, tal como hemos supuesto hasta el momento. Supongamos ahora que conocemos tanto π como la longitud de las agujas precisamente, pero que no tenemos a mano una regla para medir la

longitud de las planchas. Primero nos aseguramos de que las agujas sean más cortas que la anchura de las planchas. En este (¡improbable!) escenario, podemos estimar la anchura de las planchas con una precisión arbitraria dejando caer un número de agujas n suficientemente grande, y contando el número k de agujas que caen encima de una rendija. Nuestra estimación es, simplemente, $w \approx 2\lambda n/k\pi$. Como siempre, cuantas más agujas tiremos, más precisa será nuestra estimación.

Con una probabilidad elevada, los algoritmos para estimar el valor de π y la anchura de las planchas proporcionan un valor que converge a la respuesta correcta a medida que el número de agujas tiradas tiende a infinito (siempre y cuando las agujas tengan una longitud igual a la mitad de las planchas en el primer caso, y conociendo con precisión π y la longitud de las agujas en el segundo). La pregunta natural que se hace uno es: ¿con qué velocidad convergen estos algoritmos? Lamentablemente, son muy lentos: se necesita dejar caer la aguja 100 veces más que hasta el momento para obtener un dígito decimal adicional de precisión.

El análisis de la convergencia de los algoritmos probabilistas numéricos requiere más conocimientos de probabilidad y de estadística que los que normalmente supondríamos por parte de nuestros lectores. Sin embargo, vamos a seguir con un esbozo del enfoque básico para aquellos que tengan los conocimientos necesarios. Nos concentraremos en el análisis del algoritmo para estimar π. Consideremos un ε positivo y arbitrariamente pequeño. Asociamos una variable aleatoria X_i a cada aguja: $X_i = 1$ si la i-ésima aguja cae encima de una rendija, y $X_i = 0$ en caso contrario. Por el teorema de Buffon, $\Pr[X_i = 1] = 1/\pi$ para cada i. La esperanza y varianza de X_i son $E(X_i) = 1/\pi$ y $Var\,(X_i) = \frac{1}{\pi}(1 - \frac{1}{\pi})$, respectivamente. Ahora, supongamos que X representa la variable aleatoria correspondiente a nuestra estimación de $1/\pi$ después de dejar caer n agujas: $X = \sum_{i=1}^{n} X_i/n$. Para cualquier entero k entre 0 y n,

$$\Pr[X = k/n] = \binom{n}{k}\left(\frac{1}{\pi}\right)^k\left(1-\frac{1}{\pi}\right)^{n-k}.$$

Por ejemplo, cuando $n = 22$, $\Pr[X = \frac{7}{22}] \approx 18\%$ y $\Pr[X = \frac{6}{22} \le X \le \frac{8}{22}]$ es un poquito superior al 50%. Esta variable aleatoria tiene una esperanza $E(X) = 1/\pi$ y una varianza $Var(X) = \frac{1}{\pi}(1 - \frac{1}{\pi})/n$. Por el Teorema Central del Límite, la distribución de X es casi normal siempre y cuando n sea suficientemente grande; véase la Sección 1.7.4. Las tablas de la distribución normal nos dicen que es

$$\Pr[|X - E(X)| < 1{,}645\sqrt{Var(X)}] \approx 90\%.$$

Utilizando nuestros valores para $E(X)$ y para Var(X), inferimos que $\Pr[|X - \frac{1}{\pi}| < \varepsilon] \ge 90\%$ cuando

$$\varepsilon > 1{,}645\sqrt{\tfrac{1}{\pi}\left(1-\tfrac{1}{\pi}\right)/n}$$

y por tanto cuando $n > 0{,}5881/\varepsilon^2$. Por tanto, basta con dejar caer al menos $0{,}5881/\varepsilon^2$ agujas para obtener un error absoluto en nuestra estimación de $1/\pi$ que no sobre-

pase ε nueve veces de cada diez. Esta dependencia de $1/\varepsilon^2$ explica la razón por la cual un dígito adicional de precisión —que significa que ε tiene que ser 10 veces menor— requiere 100 veces más agujas. Si no estamos satisfechos con un intervalo de confianza que sólo resulta fiable nueve veces de cada diez, se puede utilizar otra entrada de la tabla de distribución normal. Por ejemplo, el hecho de que

$$\Pr\left[\,|X - E(X)| < 2{,}576\ \sqrt{Var(x)}\,\right] \approx 99\%$$

nos dice que $\Pr\left[\,|X - \frac{1}{\pi}| < \varepsilon\right] \geq 99$ supuesto que, $\varepsilon > 2{,}576\ \sqrt{\frac{1}{\pi}\left(1 - \frac{1}{\pi}\right)/n}$ lo cual se satisface cuando $n > 1{,}440/\varepsilon^2$. Por tanto, una reducción de diez veces en la probabilidad de error resulta poco generosa en comparación con un aumento de una cifra de precisión. En general, una vez que hayamos decidido cuántas agujas vamos a dejar caer, existe una compensación entre el número de dígitos que obtenemos y la probabilidad de que esos dígitos sean correctos.

Por supuesto, nos interesa hallar un intervalo de confianza para nuestra estimación de π, y no de $1/\pi$. Un cálculo sencillo muestra que si $\left|X - \frac{1}{\pi}\right| < \varepsilon$ entonces $\left|\frac{1}{X} - \pi\right| < \frac{\varepsilon\pi^2}{1 - \varepsilon\pi}$. Y además $9{,}8\,\varepsilon < \frac{\varepsilon\pi^2}{1 - \varepsilon\pi} < 10\varepsilon$ cuando $\varepsilon < 0{,}00415$, lo cual significa intuitivamente que un mismo número de agujas proporcionará una cifra decimal menos respecto a π que respecto a $1/\pi$, aun cuando el error relativo sea esencialmente el mismo puesto que π es aproximadamente 10 veces mayor que $1/\pi$. Resumiendo todo esto, suponiendo que necesitemos un error absoluto en nuestra estimación de π que no sobrepase $\varepsilon < 0{,}0415$ al menos el 99% de las veces, basta con dejar caer $144/\varepsilon^2$ agujas. Bastan unas pocas agujas menos en el límite de ε pequeño. ¡Esto es casi un millón y medio de agujas aunque nos conformemos con que nuestra estimación diste menos de 0,01 respecto al valor exacto de π en el 99% de las ocasiones! Nunca dijimos que esta aproximación fuera práctica, ¿verdad?

Para resumir, el usuario del algoritmo debería proporcionar dos parámetros: la fiabilidad deseada ρ, y la precisión ε. A partir de estos parámetros, con ayuda de una tabla de distribución normal, el algoritmo determina el número n de agujas que es necesario dejar caer para que la estimación resultante esté entre $\pi - \varepsilon$ y $\pi + \varepsilon$ con una probabilidad ρ como mínimo. Por ejemplo, el algoritmo seleccionaría un n próximo a un millón y medio si el usuario indicase $\rho = 99\%$ y $\varepsilon = 0{,}01$. Por último, el algoritmo deja caer n agujas al suelo, cuenta el número k de agujas que caen encima de una rendija, y proclama: «El valor de π está probablemente entre $\frac{n}{k} - \varepsilon$ y $\frac{n}{k} + \varepsilon$». Esta respuesta será correcta una fracción ρ de las veces si se repite el experimento varias veces con los mismos valores de ρ y de ε.

Queda una sutileza si se desea implementar un algoritmo similar para estimar un valor que no se conozca de antemano: ¡necesitamos el valor de π para nuestro análisis de convergencia! ¿Cómo podemos determinar el número requerido de agujas, n, como función únicamente de ρ y de ε? Una solución es ser muy conservador en nuestra estimación de la varianza de las variables aleatorias implicadas. La varianza de X_i es $p(1-p)$ cuando $X_i = 1$ con probabilidad p, y $X_i = 0$ en caso contrario, lo cual es casi un cuarto (el caso peor es aquél en el que $p = {}^1/_2$). Si determinamos n como si la varianza de X_i fuese un cuarto, entonces dejaremos caer más agujas que las estrictamente necesarias, pero la precisión deseada se obtendrá con una fiabilidad mayor que la re-

querida. Entonces el algoritmo puede estimar la fiabilidad real obtenida empleando una tabla de la distribución t de Student. Los detalles se pueden encontrar en cualquier texto estandar de estadística. Una solución alternativa es utilizar una pequeña muestra de tamaño arbitrario para obtener una primera estimación del valor requerido, utilizar esto para hacer una estimación mejor del tamaño de muestra necesario, y así sucesivamente. Una vez más, los detalles están en cualquier texto estándar.

10.5.2 Integración numérica

Esto nos lleva al más conocido de los algoritmos probabilistas numéricos: la integración de Monte Carlo, cuyo nombre es poco afortunado porque *no* es un ejemplo de algoritmo de Monte Carlo según la terminología empleada en este libro. Recordemos que si $f: \mathbb{R} \to \mathbb{R}^{\geq 0}$ es una función continua y si $a \leq b$, entonces el área de la superficie limitada por la curva $y = f(x)$, el eje x y las líneas verticales $x = a$ y $x = b$ es

$$I = \int_a^b f(x)dx$$

Consideremos ahora un rectángulo de anchura $b - a$ y de altura $I/(b - a)$ situado tal como indica la figura 10.2. El área del rectángulo también es I. Dado que el rectángulo y la superficie bajo la curva tienen la misma área, tienen que tener la misma altura media. Concluimos que la altura media de la curva entre a y b es $I/(b - a)$. Si consideramos que los puntos que están bajo el eje X tienen altura negativa, entonces esta interpretación es válida más generalmente para funciones continuas $f: \mathbb{R} \to \mathbb{R}$.

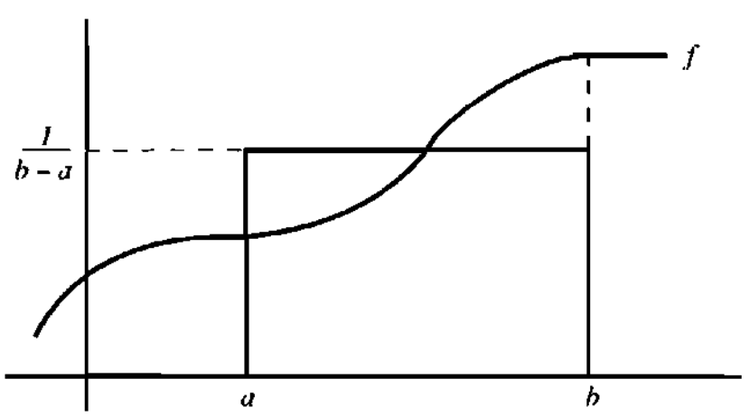


Figura 10.2 Integración numérica

Después de leer la sección anterior, un algoritmo probabilista para estimar la integral debería ocurrírsenos inmediatamente: estimar la altura media de la curva por muestreo aleatorio y multiplicar el resultado por $b–a$.

```
función intMC(f, n, a, b)
    suma ← 0
    para i ← 1 hasta n hacer
        x ← uniforme(a, b)
        suma ← suma + f(x)
    devolver (b-a) × (suma/n)
```

Un análisis similar al de la aguja de Buffon muestra que la varianza de la estimación calculada mediante este algoritmo es inversamente proporcional al número de puntos de la muestra, y que la distribución de la estimación es aproximadamente normal cuando n es muy grande. Por tanto, el error esperado en la estimación es de nuevo inversamente proporcional a $\sqrt{n}$, así que se necesita 100 veces más trabajo para obtener un dígito adicional de precisión.

El algoritmo *intMC* es apenas más práctico que el método de Buffon para estimar π; véase el problema 10.3. En general, se puede obtener una estimación mejor de la integral mediante métodos deterministas. Quizá el más sencillo es de espíritu similar a *intMC*, salvo que estima la altura media por muestreo determinista a intervalos regulares:

```
función intDET(f, n, a, b)
    suma ← 0
    delta ← (b − a)/n
    x ← a + delta/2
    para i ← 1 hasta n hacer
        suma ← suma + f(x)
        x ← x + delta
    devolver suma × delta
```

En general, este algoritmo determinista necesita muchas menos iteraciones que la integración de Monte Carlo para obtener un grado de precisión comparable. Esto es típico de la mayoría de las funciones que podamos desear integrar. Sin embargo, a todo algoritmo determinista de integración, incluso al más sofisticado, le corresponden funciones continuas diseñadas expresamente para engañar al algoritmo. Considérese por ejemplo la función $f(x) = \sin^2((100!)\pi x)$. Toda llamada a *intDET*$(f, n, 0, 1)$ con $1 \leq n \leq 100$ devuelve el valor 0, aun cuando el valor real de esta integral es $1/2$. Ninguna función puede hacerle esta jugada al algoritmo de Monte Carlo, aunque existe una probabilidad extremadamente pequeña de que el algoritmo pudiera cometer un error similar aun cuando f sea una función completamente normal.

Una razón mejor para utilizar la integración de Monte Carlo surge en la práctica cuando se tiene que evaluar una integral múltiple. Si se generaliza un algoritmo de integración determinista que utilice algún método sistemático para muestrear la función, adecuándolo a varias dimensiones, entonces el número de puntos de muestra necesarios para lograr una precisión dada crece de manera exponencial con la dimensión de la integral que haya que evaluar. Si se necesitan 100 puntos para evaluar una integral simple, entonces es probable que sea necesario utilizar los 10.000 puntos de una retícula 100×100 para alcanzar la misma precisión cuando se evalúe una integral doble; se necesitará un millón de puntos para una integral triple, y así sucesivamente. Por otra parte, empleando la integración de Monte Carlo, la dimensión de la integral suele tener poco efecto sobre la precisión obtenida, aunque es probable que la cantidad de trabajo para cada iteración aumente ligeramente con la dimensión. En la práctica, la integración de Monte Car-

lo se utiliza para evaluar integrales de cuatro o más dimensiones, porque no hay ninguna otra técnica sencilla que sea competitiva. Sin embargo, hay algoritmos mejores, que son más complicados. Por ejemplo, se puede mejorar la precisión de la respuesta empleando técnicas híbridas que son parcialmente sistemáticas y parcialmente probabilistas. Si la dimensión es fija, puede ser preferible seleccionar sistemáticamente los puntos de tal modo que estén igualmente espaciados y sea razonable su número; esta técnica se conoce con el nombre de *cuasi* Monte Carlo.

10.5.3 Conteo probabilista

¿Hasta dónde se puede contar con un registro de n bits? Empleando la notación binaria ordinaria, se puede contar desde 0 hasta $2^n - 1$. ¿Se puede ir más allá? Claramente, el registro no puede suponer más de 2^n valores distintos, puesto que ése es el número de formas distintas de n cifras binarias. Sin embargo, podríamos ir más allá de 2^n si estuviéramos dispuestos a descartar valores intermedios. Por ejemplo, se podría contar el doble si estuviéramos dispuestos a contar solamente los valores pares. Más exactamente, sea c nuestro registro de n *bits*. Algunas veces utilizaremos también c para denotar el entero que representa el registro en binario. Deseamos implementar dos procedimientos *iniciar*(c) y *pulsar*(c), y una función *contar*(c) tal que una llamada a *contar*(c) devuelva el número de llamadas a *pulsar*(c) habidas desde la última llamada a *iniciar*(c). En otras palabras, *iniciar* pone el contador a cero, *pulsar* le suma 1 y *contar* pide su valor actual. Los algoritmos deben ser capaces de mantener un número arbitrariamente grande de contadores c_1, c_2, ..., y no se admiten los efectos secundarios: no se puede pasar información entre llamadas, salvo a través del registro que se transmite como parámetro explícito.

Sugeríamos más arriba que se podían saltar algunos valores para llegar más lejos. Sin embargo, esto carece de sentido si insistimos en utilizar una estrategia *determinista* de recuento. Dado que no se admiten efectos secundarios, no hay forma de que *pulsar* pueda sumar 1 al contador en llamadas alternativas: el comportamiento de *pulsar*(c) debe de estar completamente determinado a partir del valor actual de c. Si existe un valor de c tal que *pulsar*(c) deja intacto a c, entonces el contador se detendrá en ese punto hasta que se llame a *iniciar*(c). (Esto es razonable cuando el contador ha alcanzado su límite superior.) Dado que c sólo puede admitir 2^n valores diferentes, el contador se ve obligado a tomar un valor que ya haya tenido anteriormente al cabo de un máximo de 2^n llamadas a *pulsar*(c). Por tanto, es imposible contar de manera determinista más de 2^n sucesos en un registro de n *bits*.

Si relajamos el requisito consistente en que *contar* debe de devolver el número exacto de llamadas a *pulsar* desde el último *iniciar*, existe una estrategia probabilista evidente para contar dos veces más. El registro es un contador binario normal, que se pone a cero mediante una llamada a *iniciar*. Cada vez que se llama a *pulsar*, lanzamos al aire una moneda imparcial. Si sale cara, sumamos 1 al registro. Si sale cruz, sin embargo, no hacemos nada. Cuando se llama a *contar*, devuelve el doble del valor almacenado en el registro. Después de una llamada a *iniciar*, es fácil demostrar que el valor esperado que proporciona *contar* después de t llamadas

a *pulsar* es precisamente t, aun cuando t sea impar. Se puede demostrar que la varianza es razonablemente pequeña, así que la estimación proporcionada por *contar* es bastante precisa en la mayoría de las ocasiones; véase el problema 10.5.

No esperamos impresionarle con el párrafo anterior. Contar dos veces más significa simplemente llegar hasta $2^{n+1}-2$, lo cual se podría haber logrado de forma determinista si hubiera estado disponible un solo *bit* más dentro del registro. Ahora mostraremos que las estrategias de conteo probabilista pueden llegar *exponencialmente* más allá: desde 0 hasta $2^{2^c-1}-1$. Por tanto, bastan 8 *bits* para contar ¡más de 5×10^{76} sucesos! La idea consiste en mantener en el registro una estimación del logaritmo del número real de llamadas a pulsar. Más exactamente, *contar*(c) devuelve $2^c - 1$. Le restamos 1 para que se pueda representar un conteo de 0; además *contar*(0) = 0, tal como debe de ser, porque *iniciar*(c) pone c a 0. Queda por ver la forma en que se implementa *pulsar*.

Supongamos que $2^c - 1$ es una buena estimación del número de llamadas a pulsar desde la última iniciación. Después de un adicional a pulsar, una buena estimación del número total de pulsos sería 2^c, pero esto no se puede representar así con nuestra estrategia. Para soslayar el problema, sumaremos 1 a c con alguna probabilidad p que ya se determinará, y dejaremos c intacto en caso contrario. De esta manera, nuestra estimación del número de llamadas a pulsar pasa a ser $2^{c+1} - 1$ con probabilidad p, mientras que tiene una probabilidad $1 - p$ la posibilidad de que el contador persista con su valor $2^c - 1$. El valor esperado que devuelve *contar*(c) después de esta llamada a pulsar es por tanto

$$\left(2^{c+1}-1\right)p + \left(2^c - 1\right)(1-p) = 2^c + 2^c p - 1$$

y se ve que basta hacer $p = 2^{-c}$ para obtener 2^c, que es el valor esperado que se deseaba para *contar*(c). (Este razonamiento no es riguroso; hacer el problema 10.6 para una demostración de que el valor esperado que devuelve *contar* es el correcto.) Para resumir, obtenemos los algoritmos siguientes:

```
procedimiento iniciar(c)
    c ← 0

procedimiento pulsar(c)
    para 1 ← 1 hasta n hacer
        si tiramoneda = cara entonces terminar
    c ← c + 1
    {La probabilidad de desbordamiento es demasiado pequeña para que valga la
                                                        pena comprobarlo }
función contar(c)
    devolver 2^c - 1
```

Al guardar el logaritmo del conteo actual en c, se obtiene una ventaja adicional. Vimos en las secciones anteriores que los algoritmos probabilistas numéricos mantenían bajo control el error absoluto. Cuando se implementa un contador cuyo

rango alcance es tan grande como 5×10^{76}, es más probable que se necesite mantener bajo control el error *relativo*. Afortunadamente, un pequeño error absoluto en el logaritmo del número de llamadas a pulsar se traduce en un pequeños error relativo del contador real. En particular, se garantiza que *contar* dará la respuesta exacta cuando el número de pulsos habidos desde la inicialización es 0 o 1. Se puede probar por inducción matemática que la varianza de *contar* al cabo de m llamadas a pulsar es $m(m-1)/2$. Esto no está demasiado bien, porque significa que la desviación estándar es alrededor del 70% del valor real de conteo: este proceso distinguiría un millón de mil millones, pero no sería fiable para distinguir un millón de dos millones.

Aunque hay aplicaciones para las cuales una carencia de precisión semejante resulta aceptable, ninguna aplicación real puede tener necesidad de contar hasta 5×10^{76} sucesos. Afortunadamente, la varianza del conteo estimado se puede mejorar a costa de no llegar tan lejos al contar. Para esto, basta llevar la cuenta en el registro del logaritmo en una base que sea menor que dos. Por ejemplo, podríamos utilizar

$$contar(c) = \left[(1+\varepsilon)^c - 1\right] / \varepsilon \qquad (10.1)$$

para ε pequeño. La división por ε es para mantener la propiedad deseable de que *contar* siempre da la respuesta exacta al cabo de 0 o 1 llamadas a pulsar. Con $\varepsilon = {}^1/_{30}$, esto nos permite contar hasta más de 125.000 sucesos en un registro de 8 *bits*, con un error relativo menor que el 24% durante el 95% del tiempo. Por supuesto, la probabilidad de que *pulsar* vaya a incrementar c cuando sea invocado ya no debe de ser 2^{-c}: hay que darle un valor adecuado; véase el problema 10.8.

Para un tipo de conteo probabilista completamente diferente que ahorra tiempo en lugar de espacio, asegúrese de hacer el problema 10.12. En ciertas condiciones, este algoritmo puede contar aproximadamente el número de elementos que hay en un conjunto en un tiempo que esta en el orden de la raíz cuadrada de ese número.

10.6 ALGORITMOS DE MONTE CARLO

Existen problemas para los cuales no se conoce ningún algoritmo eficiente (probabilista o determinista) que sea capaz de obtener una solución correcta en todas las ocasiones. Los algoritmos de *Monte Carlo* cometen ocasionalmente un error, pero encuentran la solución correcta con una probabilidad alta sea cual sea el caso considerado. Esto es una afirmación más fuerte que decir que funciona correctamente en la mayoría de los casos, fallando tan sólo de vez en cuando en algunos casos especiales: no debe haber ningún caso en el cual la probabilidad de error sea elevada. Sin embargo, no suele darse ningún aviso cuando el algoritmo comete un error.

Sea p un número real tal que $0 < p < 1$. Decimos que un algoritmo de Monte Carlo es *p-correcto* si devuelve una respuesta correcta con probabilidad no menor que p, sea cual sea el caso considerado. En algunos casos, permitiremos que p dependa del tamaño del caso, pero nunca del caso en sí. La característica más importante de los algoritmos de Monte Carlo es que suele ser posible reducir arbitrariamente la probabilidad de error a costa de un ligero aumento del tiempo de cálculo. Llamamos a esto *amplificación de la ventaja estocástica*. Estudiaremos este fenómeno general en la Sección 10.6.4, pero primero consideramos dos ejemplos. El primero tiene escaso interés práctico, pero tiene la ventaja de su sencillez. Lo utilizamos para presentar las nociones clave. El segundo, por otra parte, tiene un indudable interés práctico.

10.6.1 Verificación de la multiplicación de matrices

Consideremos tres matrices $n \times n$, A, B y C. Se sospecha que $C = AB$. ¿Cómo se puede verificar si esto es o no verdad? La aproximación evidente consiste en multiplicar A por B, y comparar el resultado con C. Empleando un sencillo algoritmo de multiplicación matricial, esto requiere un tiempo que está en $\Theta(n^3)$. El algoritmo de Strassen es más rápido, y existen algoritmos de multiplicación aún más rápidos para valores muy grandes de n, pero el algoritmo asintóticamente más rápido conocido hasta el momento sigue requiriendo un tiempo que está en $\Omega(n^{2,37})$; véase la Sección 7.6. ¿Podremos hacerlo mejor si estamos dispuestos a admitir una pequeña probabilidad de error? Sorprendentemente, la respuesta es que para cualquier $\varepsilon > 0$ prefijado, basta un tiempo que está en $O(n^2)$ para resolver este problema con una probabilidad de error que sea como mucho ε. La constante oculta en la notación O, sin embargo, depende de ε.

Supongamos que $C \neq AB$. Sea $D = AB - C$. Por hipótesis, D no es idénticamente nulo. Sea i un entero tal que la i-ésima fila de D contiene al menos un elemento no nula. Consideremos cualquier subconjunto $S \subseteq \{1, 2, \ldots, n\}$. Supongamos que $\Sigma_S(D)$ representa el vector de longitud n que se obtiene sumando componente a componente las filas de D indexadas por los elementos de S; una suma vacía es igual a cero. Por ejemplo, $\Sigma_{\{1, 3, 4\}}(D)$ se obtiene sumando las filas primera, tercera y cuarta de D. Sea S' el mismo subconjunto que S, salvo que $i \in S'$ si y sólo si $i \notin S$. Dado que la i-ésima fila de D no es **idénticamente** 0, $\Sigma_S(D)$ y $\Sigma_{S'}(D)$ no pueden ser nulos simultáneamente. Supongamos ahora que seleccionamos S de forma aleatoria: para todo j entre 1 y n, decidimos mediante una moneda imparcial si se incluye o no j en S. Dado que es tan probable que $i \in S$ como que $i \notin S$, la probabilidad de que $\Sigma_S(D) \neq 0$ es como mínimo de un medio.

Por otra parte, $\Sigma_S(D)$ siempre es cero si $C = AB$, puesto que en este caso $D = 0$. Esto sugiere comprobar si $C = AB$ o no calculando $\Sigma_S(D)$ para un subconjunto S seleccionado aleatoriamente, y comparando con 0 el resultado. Pero ¿cómo se puede hacer esto eficientemente sin calcular primero D (y por tanto AB)?

La solución consiste en considerar $\Sigma_S(D)$ como una multiplicación de matrices independiente. Sea X un vector binario de longitud n tal que $X_i = 1$ si $j \in S$ y $X_i = 0$ en caso contrario. Es sencillo verificar que $\Sigma_S(D) = XD$, siempre que pensemos en X como en una matriz $1 \times n$. De esta manera, nuestra comprobación se corresponde con verificar si $X(AB - C)$ es o no cero, o equivalentemente si $XAB = XC$, para algún vector binario X seleccionado arbitrariamente. A primera vista, parece que calcular XAB debiera de requerir todavía más tiempo que calcular AB, pero esto no es así. Recordemos de la Sección 8.6 que el tiempo necesario para calcular una multiplicación matricial encadenada depende crucialmente del orden en que se multiplican las matrices. En este caso, sólo se necesita un tiempo que está en $\Theta(n^2)$ para calcular XAB en la forma $(XA)B$, porque éste es el tiempo que se necesita para multiplicar la matriz $1 \times n$ X por la matriz $n \times n$ A, y porque el resultado XC es de nuevo una matriz $1 \times n$ que podemos multiplicar por B en un tiempo que está en $\Theta(n^2)$. Esto sugiere el algoritmo siguiente:

```
función Freivalds(A, B, C, n)
    para j ← 1 hasta n hacer X_j ← uniforme (0..1)
    si (XA)B = XC entonces devover verdadero
                sino devolverr falso
```

A la vista de lo dicho anteriormente, sabemos que este algoritmo devuelve una respuesta correcta con una probabilidad del 50% en todos los casos, por tanto, es 1/2-correcto por definición. Sin embargo no es p-correcto para todo p menor que 1/2, porque su probabilidad de error es exactamente 1/2 cuando C difiere de AB precisamente en una fila: se devuelve una respuesta incorrecta si y sólo si $X_i = 0$ donde la fila i supone la diferencia entre C y AB. ¿Es esto realmente interesante? Una probabilidad de error del 50% es intolerable en la práctica, ¿verdad? Hay algo peor: un «algoritmo» más sencillo para alcanzar esta probabilidad de error sería arrojar al aire una moneda imparcial y ¡*devolver verdadero* o *falso* dependiendo del resultado, sin siquiera examinar las tres matrices en cuestión! La observación clave es que siempre que *Freivalds*(A, B, C, n) devuelve el valor *falso*, uno puede estar seguro de que la respuesta es correcta: la existencia de un vector X tal que XAB difiere de XC nos permite concluir con certeza que $AB \neq C$. Sólo cuando el algoritmo devuelve *verdadero* estamos inseguros a la hora de creer su afirmación.

Consideremos por ejemplo las matrices 3×3 siguientes:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{pmatrix} \quad C = \begin{pmatrix} 11 & 29 & 37 \\ 29 & 65 & 91 \\ 47 & 99 & 45 \end{pmatrix}$$

Una llamada a *Freivalds*$(A, B, C, 3)$ podría seleccionar $X = (1, 1, 0)$ en cuyo caso XA se obtiene sumando las filas primera y segunda de A, así que $XA = (5, 7, 9)$. Prosiguiendo, calculamos $(XA)B = (40, 94, 128)$. Este resultado se compara con $XC = (40,$

94, 128), que se obtiene sumando la primera y segunda fila de C. Con esta selección de X, *Freivalds* devuelve el valor *verdadero* puesto que es $XAB = XC$. Otra llamada a *Freivalds* podría seleccionar $X = (0, 1, 1)$. En este caso, los cálculos son

$$XA = (11, 13, 15),\ (XA)B = (76, 166, 236),\ XC = (76, 164, 136)$$

y *Freivalds* proporciona el valor *falso*. Tenemos más suerte: el hecho de que $XAB \neq XC$ es una demostración concluyente de que $AB \neq C$.

Esto sugiere aplicar *Freivalds*(A, B, C, n) varias veces sobre el mismo caso, empleando tiradas independientes de la moneda en cada ocasión. Si se obtiene la respuesta *falso*, incluso en una sola ocasión, podemos concluir que $AB \neq C$ independientemente del número de veces que se obtenga la respuesta *verdadero*. Consideremos ahora el siguiente algoritmo, en el cual k es un nuevo parámetro.

```
función RepetirFreivalds(A, B, C, n, k)
    para i ← 1 hasta k hacer
        si Freivalds(A, B, C, n) = falso entonces devolver falso
    devolver verdadero
```

Nos interesa la probabilidad de error de este nuevo algoritmo. Consideremos dos casos. Si realmente $AB = C$, entonces toda llamada a *Freivalds* devuelve necesariamente el valor *verdadero*, puesto que no existe la posibilidad de seleccionar aleatoriamente un X tal que $XAB \neq XC$, y por tanto *RepetirFreivalds* devuelve *verdadero* después de pasar k veces por el bucle. En este caso, la probabilidad de error es cero. Por otra parte, si $C \neq AB$, entonces la probabilidad de que una llamada a *Freivalds* devuelva (incorrectamente) *verdadero* es como máximo 1/2. Estas probabilidades se multiplican, porque los lanzamientos de la moneda son independientes entre llamadas consecutivas. Por tanto, la probabilidad de que k llamadas consecutivas devuelvan todas ellas pueden devolver un valor erróneo de 2^{-k} como máximo. Pero ésta es la única forma en que *RepetirFreivalds* devuelve una respuesta errónea. Concluimos que nuestro nuevo algoritmo $(1 - 2^{-k})$-correcto. Cuando $k = 10$, esto es mejor que 99,9%-correcto. Repetirlo 20 veces lleva la probabilidad de error a menos de una entre un millón. Este decrecimiento espectacularmente rápido de la probabilidad de error es típico de los algoritmos de Monte Carlo que resuelven problemas de decisión —esto es, problemas para los cuales la respuesta es o bien *verdadero* o bien *falso*— siempre y cuando una de las respuestas, en caso de ser obtenida, tenga garantizada su veracidad.

Alternativamente, se puede dar a los algoritmos de Monte Carlo una cota superior explícita para la probabilidad de error admisible:

```
función FreivaldsEpsilon(A, B, C, n, ε)
    k ← ⌈lg 1/ε⌉
    devolver RepetirFreivalds(A, B, C, n, k)
```

Una ventaja de esta versión del algoritmo es que podemos analizar su tiempo de ejecución como función simultánea del tamaño del caso y de la probabilidad de error. En este caso, el algoritmo requiere claramente un tiempo que está en

$\Theta(n^2 \log^1/_\epsilon)$.

Los algoritmos de esta sección tienen un interés práctico limitado, porque se necesitan $3n^2$ multiplicaciones escalares para calcular XAB y XC, en comparación con las n^3 que se necesitan para calcular AB por el método directo. Si insistimos en una probabilidad de error que no sea mayor que una entre un millón, y si de hecho $C = AB$, entonces las 20 ejecuciones necesarias de *Freivalds* efectúan $60n^2$ multiplicaciones escalares, lo cual no es una mejora con respecto a n^3 a no ser que n sea mayor que 60. Sin embargo, esta aproximación podría resultar de utilidad si fuera necesario verificar productos matriciales muy grandes.

10.6.2 Comprobación de primalidad[2]

Es posible que el algoritmo de Monte Carlo más famoso sea el que decide si un entero impar dado es o no primo. No se conoce ningún algoritmo para resolver este problema con certeza en un tiempo razonable cuando el número que hay que estudiar no tiene más que unos pocos centenares de dígitos decimales. Comprobar la *primalidad* de números grandes es algo más que un *divertimento* matemático: ya vimos en la Sección 7.8 que es crucial para la criptología moderna.

La historia de la comprobación probabilista de *primalidad* tiene sus raíces en Pierre de Fermat, el padre de la Teoría de los Números moderna. En 1640 enunció el teorema siguiente, que a veces se denomina teorema menor de Fermat.

> Teorema10.6.1 (Fermat). Sea n un primo. Entonces a^{n-1} mód$n = 1$ para cualquier entero a tal que $1 \le a \le n - 1$.

Por ejemplo, sean $n = 7$ y $a = 5$. Se tiene que es

$$a^{n-1} = 5^6 = 15625 = 2232 \times 7 + 1$$

y por tanto es $a^{n-1} \bmod n = 1$, tal como debía de suceder. Considérese ahora la versión contrapositiva del teorema de Fermat: si n y a son enteros tales que $1 \le a \le n-1$ y si $a^{n-1} \bmod n \neq 1$ entonces n no es primo. Un ejemplo anecdótico lo proporcionó el mismo Fermat. En su búsqueda de una fórmula que sólo produjese números primos, formuló la hipótesis de que $F_n = 2^{2^n} + 1$ es primo para todo n. Obsérvese que $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$ y $F_4 = 65537$ son todos ellos números primos. Desafortunadamente, $F_5 = 4294967297$ era un número demasiado grande para que Fermat pudiera intentar demostrar su primalidad. Tuvo que pasar casi un siglo para que Euler factorizase $F_5 = 641 \times 6700417$, probando así la falsedad de la conjetura de Fermat. Irónicamente, Fermat podría haber utilizado su propio teorema para alcanzar la misma conclusión. Estaba dentro de sus posibilidades inventar un algoritmo eficiente de exponenciación modular parecido a *expomod*, y utilizarlo

para calcular $3^{F_5-1} \bmod F_5 = 3029026160 \neq 1$; véase la Sección 7.8. Aunque es tedioso, este cálculo no requiere nada más que 32 cuadrados modulares sucesivos, puesto que $F_5 - 1 = 2^{32}$. A partir de la versión contrapositiva de su propio teorema, Fermat podría haber sabido que F_5 no es primo.

Esto sugiere el siguiente algoritmo probabilista para la comprobación de *primalidad*. Suponemos que $n \geq 2$:

```
función Fermat(n)
    a ← uniforme(1..n – 1)
    si expomod(a, n – 1, n) = 1 entonces devolver verdadero
    sino devolver falso
```

Visto lo anterior, sabemos que n es compuesto cuando una llamada a *Fermat*(n) devuelva el valor *falso* porque la existencia de un a entre 1 y $n-1$ tal que $a^{n-1} \bmod n \neq 1$ basta para demostrar esto, por el teorema menor de Fermat. Es notable que cuando este algoritmo, indica que n es compuesto, no ofrece ni una sola indicación acerca de sus divisores. Teniendo en cuenta los conocimientos algorítmicos actuales, la factorización es mucho más difícil que la comprobación de primalidad. Recuerde que la supuesta diferencia de dificultad entre estos problemas es crucial para el sistema criptográfico RSA descrito en la Sección 7.8: la facilidad de comprobar la primalidad es necesaria para su implementación, mientras que la dificultad de la factorización es condición indispensable para su seguridad.

¿Qué se puede decir, sin embargo, si *Fermat*(n) devuelve el valor *verdadero*? Para concluir que n es primo, necesitaríamos el recíproco y el contrapositivo del teorema de Fermat. Este resultado diría que $a^{n-1} \bmod n$ nunca es igual a 1 cuando n es compuesto y $1 \leq a \leq n-1$. Desafortunadamente, esto no es cierto, porque $1^{n-1} \bmod n = 1$ para todo $n \geq 2$. Además, $(n-1)^{n-1} \bmod n = 1$ para todo $n \geq 3$, porque $(n-1)^2 \bmod n = 1$. El menor contraejemplo no trivial del recíproco es que $4^{14} \bmod 15 = 1$ a pesar de que 15 sea compuesto. Un entero a tal que $2 \leq a \leq n-2$ y que $a^{n-1} \bmod n = 1$ se denomina *testigo falso de primalidad* para n si de hecho n es compuesto. Con tal de modificar la comprobación de Fermat para seleccionar aleatoriamente a entre 2 y $n-2$, sólo fallará en los números compuestos cuando seleccionemos un testigo falso.

Las buenas noticias son que los testigos falsos son bastante escasos. Aunque sólo 5 de entre los 332 impares compuestos menores que 1.000 presumen de no tener falsos testigos, más de la mitad tiene sólo dos testigos falsos y menos del 16% de ellos tiene más de 15. Además, sólo existen 4.490 testigos falsos para todos estos números compuestos juntos. Comparemos esta cantidad con 178.878, que es el número total de testigos candidatos que existe para el mismo conjunto de números. La probabilidad media de error para la comprobación de Fermat aplicada a números impares compuestos menores que 1.000 es menor que 3,3%. Resulta aún menor si consideramos números más grandes.

Las malas noticias son que hay números compuestos que admiten una proporción significativa de falsos testigos . El caso peor entre los números compuestos menores que 1.000 es 561, que admite 318 testigos falsos: son más de la mitad de los testigos po-

tenciales. Un caso más convincente es el que plantea un número de 15 dígitos: *Format*(651693055693681) devuelve *verdadero* con una probabilidad mayor que 99,9965% ¡a pesar de que este número es compuesto! Con más generalidad, para cualquier $\delta > 0$ existe un número infinito de números compuestos para los cuales la comprobación de Fermat descubre que son compuestos con una probabilidad menor que δ. En otras palabras, la comprobación de Fermat no es p-correcta para ningún $p > 0$ que fijemos. Consiguientemente, la probabilidad de error no se puede reducir por debajo de un ε arbitrariamente pequeño por repetición de la comprobación de Fermat durante un cierto número prefijado de veces, según la técnica que veíamos en la sección anterior.

Afortunadamente, una pequeña modificación de la comprobación de Fermat resuelve esta dificultad. Sea n un entero impar mayor que 4, y sean s y t enteros tales que $n - 1 = 2^s t$, donde t es impar. Obsérvese que $s > 0$ puesto que $n-1$ es par. Sea $B(n)$ el conjunto formado por estos enteros, definido en la forma siguiente: $a \in B(n)$ si y sólo si $2 \le a \le n-2$ y

◊ $a^t \bmod n = 1$, o bien
◊ existe un entero i, $0 \le i < s$, tal que $a^{2^i t} \bmod n = n - 1$

Esto puede parecer complicado, pero es sencillo implementarlo eficientemente. Siempre y cuando n sea impar y $2 \le a \le n - 2$, una llamada a *pruebaB*(a, n) devuelva el valor *verdadero* si y sólo si $a \in B(n)$.

```
función pruebaB(a, n)
    s ← 0; t ← n–1
    repetir
    s ← s + 1; t ← t ÷ 2;
    hasta que t mód 2 = 1
    x ← expomod(a, t, n)
    si x = 1 o x = n–1 entonces devolver verdadero
    para i ← 1 hasta s–1 hacer
     x ← x^2 mod n
    si x = n–1 entonces devolver verdadero
devolver falso
```

Por ejemplo, veamos si 158 pertenece a $B(289)$. Hacemos $s = 5$ y $t = 9$ porque $n - 1 = 288 = 2^5 \times 9$. Entonces calculamos

$$x = a^t \bmod n = 158^9 \bmod 289 = 131$$

No concluye aquí la prueba, porque 131 no es 1 es $n - -1 = 288$. A continuación, elevamos x al cuadrado (módulo n) hasta 4 veces ($s-1 = 4$) para ver si obtenemos 288.

$$a^{2^i t} \bmod n = 131^2 \bmod 289 = 110$$

$$a^{2^2 t} \bmod n = 110^2 \bmod 289 = 251$$

$$a^{2^3 t} \bmod n = 251^2 \bmod 289 = 288$$

Llegados aquí, nos detenemos porque hemos encontrado que es $a^{2^i} \bmod n = n - 1$ para $i=3 < s$, y concluimos que $158 \in B(298)$.

Una extensión del teorema de Fermat demuestra que $a \in B(n)$ para todo $2 \le a \le n-2$ cuando n es primo. Por otra parte, se dice que n es un *pseudoprimo fuerte* con base a y que a es un *testigo falso fuerte* de primalidad para n siempre que $n > 4$ es un número impar compuesto y $a \in B(n)$. Por ejemplo, ya hemos visto que 158 es un testigo falso fuerte de primalidad para 289 puesto que $289 = 17^2$ es compuesto. Esto proporciona una comprobación mejor que la de Fermat porque los testigos falsos fuertes son automáticamente testigos falsos respecto a la prueba de Fermat, pero no a la inversa. De hecho, los testigos falsos fuertes son mucho más escasos que los testigos falsos. Por ejemplo, 4 es un testigo falso de primalidad para 15, pero no es un testigo falso fuerte porque $4^7 \bmod 15 = 4$. También vimos que el 561 admite 318 falsos testigos, pero sólo 8 de éstos son testigos falsos fuertes. Considerando todos los impares compuestos menores que 1.000, la probabilidad media de seleccionar aleatoriamente un testigo falso fuerte es menor que el 1%; más del 72% de estos números compuestos ni siquiera admite un único falso testigo fuerte. La superioridad de la prueba fuerte queda ilustrada de la mejor manera posible por el hecho consistente en que todos los enteros compuestos entre 5 y 10^{13} no son pseudoprimos fuertes en al menos una de las bases 2, 3, 5, 7 o 61. En otras palabras, cinco llamadas a *pruebaB* bastan para decidir de *forma determinista* acerca de la primalidad de cualquier entero hasta 10^{13}. Hay algo más importante: a diferencia de la comprobación de Fermat, se garantiza que la proporción de testigos falsos fuertes es pequeña *para todo número impar compuesto*. Más precisamente, tenemos el teorema siguiente.

Teorema 10.6.2. Consideremos un número impar arbitrario $n > 4$:
◊ Si n es primo, entonces $B(n) = \{a \mid 2 \le a \le n-2\}$
◊ Si n es compuesto, entonces $|B(n)| \le (n-9)/4$

Consiguientemente, *pruebaB*(a, n) siempre devuelve el valor *verdadero* cuando n es un primo mayor que 4, y $2 \le a \le n-2$, mientras que devuelve el valor *falso* con una probabilidad de más de 3/4 cuando n es un entero impar compuesto mayor que 4 y a se selecciona aleatoriamente entre 2 y $n-2$. En otras palabras, el siguiente algoritmo de Monte Carlo es 3/4-correcto para la comprobación de primalidad; se conoce con el nombre de comprobación de Miller-Rabin:

```
función MillRab(n)
    {Este algoritmo sólo debe invocarse si n > 4 es impar}
    a ← uniforme(2..n-2)
    devolver pruebaB(a, n)
```

Dado que se garantiza que la respuesta *falso* sea correcta, la técnica que vimos en la sección anterior es aplicable para reducir muy rápidamente la probabilidad de error.

```
función RepetirMillRab(n, k)
    {Este algoritmo sólo debe invocarse si n > 4 es impar}
    para i ← 1 hasta k hacer
        si MillRab(n) = falso entonces devolver falso
devolver verdadero
```

Este algoritmo siempre devuelve la respuesta correcta cuando $n > 4$ es primo. Cuando $n > 4$ es un impar compuesto, cada llamada a *MillRab* tiene como mucho una probabilidad de 1/4 de llegar a un testigo falso fuerte, devolviendo *verdadero* de manera errónea. Dado que la única manera de que *RepetirMillRab* devuelva el valor *verdadero* en este caso es encontrar aleatoriamente k testigos falsos fuertes seguidos, esto sucede con una probabilidad que es como mucho de 4^{-k}. Por ejemplo, basta con tomar $k = 10$ para reducir la probabilidad a menos de una entre un millón. En conclusión, *RepetirMillRab*(n, k) es un algoritmo de Monte Carlo $(1 - 4^{-k})$-correcto para la comprobación de *primalidad*.

¿Cuánto tiempo se necesita para decidir la *primalidad* de n con una probabilidad de error acotada por ε? Debemos repetir la comprobación de Miller-Rabin k veces, en donde es $4^{-k} \leq \varepsilon$, que equivale a $2^{2k} \geq 1/\varepsilon$. Esto se consigue mediante $k = \lceil \frac{1}{2} \lg 1/\varepsilon \rceil$. Cada llamada a *MillRab* requiere una exponenciación modular, con t como exponente, y s–1 elevaciones al cuadrado modulares. Sabemos por la Sección 7.8 que la exponenciación requiere un número de multiplicaciones y elevaciones al cuadrado modulares que está en $O(\log t)$. Contando las elevaciones al cuadrado como multiplicaciones, y dado que es $\lg n > \lg(n-1) = \lg(2^s)\, t = s + \lg t$, el tiempo requerido por una llamada a *MillRab* está dominado por un número de multiplicaciones modulares que está en $O(\log n)$. Si se efectúan éstas según el algoritmo clásico, cada una de ellas requiere un tiempo que está en $O(\log^2 n)$ porque reducimos módulo n después de cada multiplicación.

Resumiendo, el tiempo total necesario para decidir acerca de la primalidad de n con una probabilidad de error acotada por ε está por tanto en $O(\log^3 n \log 1/\varepsilon)$. Esto es enteramente razonable en la práctica para números de mil dígitos y una probabilidad de error menor que 10^{-100}.

10.6.3 ¿Puede un número ser probablemente primo?

Supongamos que se aplica diez veces la comprobación de Miller-Rabin a algún entero impar n, y que se obtiene la respuesta *verdadero* en las diez ocasiones. ¿Qué podemos deducir? Hemos visto que si n era compuesto, esto podría suceder con una probabilidad máxima de $4^{-10} = 2^{-20}$, que es menor que uno entre un millón. Puede tentarnos concluir que «n es primo, salvo por una probabilidad de error dada por 2^{-20}». Sin embargo, esta afirmación carece de sentido: ¡todo entero mayor que 1 es o bien primo o bien compuesto! Lo mejor que podemos decir es «creo que n es primo; en caso contrario he observado un fenómeno cuya probabilidad de ocurrir era menor que uno entre un millón». Además, hasta esta afirmación prudencial debe considerarse con cautela si los testigos de la comprobación se han obtenido con un generador pseudoaleatorio, en lugar de ser realmente alea-

torios: quizá n sea compuesto y no hemos conseguido hallar un testigo, pero no por tener mala suerte sino porque el generador pseudoaleatorio era defectuoso; véase la Sección 10.4. Sin embargo, este fallo es improbable si se emplea un buen generador, y no nos preocuparemos más acerca de esta posibilidad.

Dado que es imposible llegar a estar seguros de que un número dado es primo mediante este algoritmo de Monte Carlo —sólo se pueden certificar los números compuestos— y dado que los primos son relevantes a efectos criptográficos, quizá se pregunte si merece la pena arriesgarse a obtener un «primo falso» para ahorrar tiempo de cálculo. Si su vida dependiera de la primalidad de un número dado, ¿no invertiría el tiempo necesario para certificar su primalidad empleando el mejor método determinista? Digamos que se tiene que decidir entre ejecutar *RepetirMillRab*(n, 150) o emplear algún método más sofisticado, que requiere un tiempo significativamente mayor pero garantiza una respuesta correcta. ¿Cuál de estas aproximaciones tiene menos probabilidades de inducirnos a error? Tal como se discute en la Sección 10.2, el método sofisticado tiene más probabilidades de ser incorrecto, a pesar de sus resultados, porque la probabilidad de que se produzca un error de *hardware* no detectado durante su (más extenso) periodo de cálculo puede ser mayor que $4^{-150} \approx 10^{-100}$. Además, uno puede tener más confianza en que su implementación de la comprobación de Miller-Rabin esté libre de errores que en el algoritmo más complicado. Así llegamos a la paradójica conclusión consistente en que un primo probable, sea cual fuere el significado dado a esta frase, ¡puede ser más fiable que uno cuya primalidad se haya «demostrado» de forma determinista! Obsérvese sin embargo que existen métodos probabilistas que producen primos aleatorios *certificados* en un tiempo menor que el que se necesita para hallar primos probabilistas mediante el uso repetido de la prueba de Miller-Rabin. Dado que se garantiza que estos métodos van a proporcionar un número primo, pertenecen a la clase de algoritmos de Las Vegas.

La importancia de interpretar correctamente el resultado de un algoritmo de Monte Carlo se puede ilustrar muy bien si se desea *generar* un número de ℓ dígitos que sea probablemente primo, quizá con propósitos criptográficos. El algoritmo evidente consiste en seguir seleccionando enteros impares aleatorios de ℓ dígitos hasta encontrar uno que pase por un número suficiente de aplicaciones de la comprobación de Miller-Rabin. Más exactamente, considérese el siguiente algoritmo, que sólo debería de utilizarse con $\ell > 1$:

```
función primoaleatorio( ℓ, k)
  repetir
    {Seleccionar un n impar aleatoriamente entre 10^(ℓ-1) y 10^ℓ - 1}
    n → 1+2 × uniforme( 10^(ℓ-1) / 2..10^ℓ / 2 - 1 )
  hasta que RepetirMillRab(n, k)
  devolver n
```

¿Qué se puede decir acerca del resultado de este algoritmo? Ya sabemos que no se puede afirmar que el número obtenido sea primo con una probabilidad de al menos $1 - 4^{-k}$. Sin embargo, tiene sentido investigar el número medio de «números primos falsos» (un eufemismo de «números compuestos») que se pro-

ducen al ejecutar m veces el algoritmo. Resulta tentador concluir de la discusión anterior que la respuesta es «como mucho $4^{-k}m$ porque nuestra probabilidad de error en cada uno de los m números producidos es como máximo de 4^{-k}». Por ejemplo, si llamamos a *primoaleatorio*(1.000, 5) un millón de veces, esperaríamos menos de 1.000 números compuestos entre el millón de números de 1.000 cifras producidos así (porque es $4^{-5}10^6 \approx 997$). Bastaría con llamar a *primoaleatorio*(1.000, 10) en su lugar para reducir este número esperado a menos de 1, a costa de no llegar a doblar el tiempo de ejecución. Esta conclusión es correcta, pero sólo porque la probabilidad de error esperada para la comprobación de Miller-Rabin para un número impar compuesto seleccionado aleatoriamente es mucho menor que 1/4. Sin embargo, ¡el razonamiento simplificado es incorrecto!

El problema es que un número impar aleatorio muy grande tiene muchas más probabilidades de ser compuesto que de ser primo. Por tanto, una llamada a *primoaleatorio* probablemente utilizará *RepetirMillRab* para probar muchos números compuestos antes de comprobar un número primo, si esto llega a suceder. La probabilidad de error inherente en cada llamada a *RepetirMillRab* aplicándolo a números compuestos se irá acumulando. Aun cuando todas las llamadas tienen probabilidad de encontrar que su número es compuesto, la probabilidad de que una de las llamadas se equivoque —dando lugar a que *primoaleatorio* devuelva este número compuesto por error— no es despreciable si son muchos los números compuestos que se comprueban antes de hallar un número primo por casualidad. Consideremos lo que sucedería si la probabilidad de error de la prueba de Miller-Rabin fuera exactamente 1/4 en todos los compuestos impares, y si llamáramos a *primoaleatorio*(1.000, 5). En cada pasada por el bucle **repetir**, sucede una de estas tres cosas:

◊ Si el número n seleccionado aleatoriamente es primo, entonces *RepetirMillRab*(n, 5) devolverá *verdadero* con toda certeza, finalizando así el bucle; *primoaleatorio* devuelve correctamente un número primo en este caso.
◊ Si el número n seleccionado aleatoriamente es compuesto, entonces hay una probabilidad exactamente igual a $4^{-5} = 1/1024$ de que *RepetirMillRab*(n, 5) devuelva el valor *verdadero*, finalizando el bucle; *primoaleatorio* devuelve erróneamente un número compuesto en este caso.
◊ En caso contrario, el algoritmo vuelve al principio del bucle con la selección aleatoria de otro número impar de mil cifras.

La probabilidad de que un número impar aleatorio de mil cifras sea primo es aproximadamente de uno entre mil; véase el problema 10.19. Por tanto, los dos primeros casos son casi equiprobables, mientras que el tercero es con mucho el más probable. Como consecuencia, el bucle de *primoaleatorio* tiene las mismas probabilidades de acabar por cualquiera de las razones posibles, así que ¡devuelve números compuestos casi con tanta frecuencia como números primos! (Una

vez más, el rendimiento de *primoaleatorio* sería mucho mejor en la realidad porque la probabilidad de error en *MillRab* es substancialmente menor que 1/4 en la mayoría de los números compuestos.)

10.6.4 Amplificación de la ventaja estocástica

Los dos algoritmos de Monte Carlo estudiados hasta el momento tienen una propiedad útil: una de las respuestas posibles siempre es correcta cuando se obtiene. Uno garantiza que un número dado es compuesto, el otro que un producto matricial dado es incorrecto. Decimos que estos algoritmos están *sesgados*. Gracias a esta propiedad, es sencillo reducir arbitrariamente la probabilidad de error repitiendo el algoritmo un número apropiado de veces. La aparición de una sola respuesta «garantizada» basta para producir una certeza; un gran número de respuestas probabilistas idénticas, por otra parte, incrementa la confianza en que se haya obtenido la respuesta correcta. Esto se conoce con el nombre de *amplificación de la ventaja estocástica*.

Supongamos que disponemos de un algoritmo de Monte Carlo *no* sesgado cuya probabilidad de error sea no nula independientemente del caso que haya que resolver, y de la respuesta proporcionada. ¿Sigue siendo posible reducir arbitrariamente la probabilidad de error repitiendo el algoritmo? La respuesta es que depende de la probabilidad de error original. Por sencillez, vamos a concentrarnos en algoritmos que conciernen a la toma de decisiones; véase sin embargo el problema 10.22). Considere un algoritmo de Monte Carlo del cual lo único que se sabe es que es p-correcto. El primer comentario evidente es que la amplificación de la ventaja estocástica es imposible a no ser que $p > 1/2$ porque siempre existe el (inútil) algoritmo 1/2-correcto:

```
función estúpida(x)
    si tiramoneda = cara entonces devolver verdadero
                           sino devolver falso
```

cuya «ventaja» estocástica no se puede amplificar. Siempre y cuando $p \geq 1/2$, se define la *ventaja* de un algoritmo de Monte Carlo p-correcto como $p - 1/2$. Todo algoritmo de Monte Carlo cuya ventaja sea positiva se puede transformar en otro cuya probabilidad de error sea tan pequeña como deseemos. Comenzaremos por un ejemplo.

Sea MC un algoritmo de Monte Carlo no sesgado 3/4-correcto para resolver algún problema de decisión. Considérese el siguiente algoritmo, que llama tres veces a $MC(x)$ y devuelve la respuesta más frecuente:

```
función MC3(x)
    t ← MC(x); u ← MC(x); v ← MC(x);
    si t = u o t = v entonces devolver t
                       sino devolver u
```

¿Cuál es la probabilidad de error de *MC3*? Sean R y W las respuestas correcta e incorrecta respectivamente. Sabemos que t, u y v tienen una probabilidad mínima de $3/4$ de ser R, independientemente unas de otras. Supongamos por sencillez que esta probabilidad es exactamente de $3/4$, puesto que claramente el algoritmo *MC3* sería todavía mejor si la probabilidad de error de *MC* fuera menor que $1/4$. Hay ocho resultados posibles para las tres llamadas a *MC*, cuyas probabilidades se resumen en la tabla siguiente:

t	u	v	prob	*MC3*
R	R	R	27/64	R
R	R	W	9/64	R
R	W	R	9/64	R
R	W	W	3/64	W
W	R	R	9/64	R
W	R	W	3/64	W
W	W	R	3/64	W
W	W	W	1/64	W

Sumando las probabilidades asociadas a las filas 1, 2, 3 y 5 concluimos que *MC3* es correcto con una probabilidad de 27/32, que es mejor que 84%.

Más generalmente, sea *MC* un algoritmo de Monte Carlo para resolver algún problema de decisión cuya ventaja sea $\varepsilon > 0$. Consideremos el siguiente algoritmo, que llama k veces a *MC* y devuelve la respuesta más frecuente:

```
función RepetirMC(x, k)
    V, F ← 0
    para i ← 1 hasta k hacer
        si MC(x) entonces V← V + 1
                     sino F ← F + 1
    si V > F entonces devolver verdadero sino devolver falso
```

¿Cuál es la probabilidad de error de *RepetirMC*? Para averiguarlo, asociamos una variable aleatoria X_i a cada llamada a *MC*: $X_i = 1$ si se obtiene la respuesta correcta y $X_i = 0$ en caso contrario. Por suposición, $\Pr[X_i = 1] \geq 1/2 + \varepsilon$ para todo i. Supongamos por sencillez que esta probabilidad es exactamente de $1/2 + \varepsilon$ puesto que en otro caso *RepetirMC* sería incluso mejor. Suponga también que k es impar, para evitar el riesgo de un empate ($V = F$) en la votación de mayoría; véase el problema 10.23. La esperanza y la varianza de X_i son $E(X_i) = 1/2 + \varepsilon$ y $\text{Var}(X_i) = (1/2 + \varepsilon)(1/2 - \varepsilon) = 1/4 - \varepsilon^2$, respectivamente. Ahora, sea $X = \sum_{i=1}^{k} X_i$ la variable aleatoria que corresponde al número de respuestas correctas al cabo de k intentos. Para cualquier entero i entre 0 y k

$$\Pr[X = i] = \binom{k}{i}\left(\tfrac{1}{2} + \varepsilon\right)^{i}\left(\tfrac{1}{2} - \varepsilon\right)^{k-i} \tag{10.2}$$

y X tiene la esperanza $E(X) = (1/2 + \varepsilon)k$ y la varianza $Var(x) = (1/4 - \varepsilon^2)k$. Nos interesa $\Pr[X \leq k/2]$, que es la probabilidad de error de *RepetirMC*.

Dados unos valores específicos para ε y k, podemos calcular esta probabilidad mediante el algoritmo de programación dinámica dado en la Sección 8.1.2 para el campeonato mundial, o bien podemos calcularlo en la forma $\sum_{i=0}^{k+2} \Pr[X = i]$ empleando la ecuación 10.2. Mejor aún, podemos emplear la fórmula del problema 10.25. Sin embargo, resulta más cómodo utilizar el Teorema Central del Límite, que dice que la distribución de X es casi normal cuando k es grande; en la práctica, $k = 30$ ya es bastante grande. Supongamos por ejemplo que se desea una probabilidad de error menor que el 5%. Las tablas para la distribución normal nos dicen que

$$\Pr\left[X < E(X) - 1.645\sqrt{Var(X)}\right] \approx 5\%.$$

Para obtener $\Pr[X \geq k/2] < 5\%$ basta que

$$k/2 < E(X) - 1.645\sqrt{Var(X)}.$$

Empleando nuestros valores para $E(X)$ y $\mathrm{Var}(X)$, esta condición se traduce en que

$$k > 2{,}706\left(\frac{1}{4\varepsilon^2} - 1\right). \tag{10.3}$$

Por ejemplo, si $\varepsilon = 5\%$, la ecuación 10.3 nos dice que «basta» con ejecutar 269 veces un algoritmo de Monte Carlo no sesgado 55%-correcto y tomar la respuesta más frecuente para obtener un algoritmo 95%-correcto. (La ecuación 10.3 da $k > 267{,}894$ pero hemos tomado $k = 269$ en lugar de $k = 268$ porque deseábamos que k fuera impar; véase el problema 10.23). En otras palabras, esta repetición traduce una ventaja de un 5% en una probabilidad de error de un 5%. Un cálculo exacto muestra que la probabilidad de error resultante está justo por encima del 4,99%, ilustrando la precisión de la aproximación normal para valores tan grandes de k. Esto muestra el inconveniente de los algoritmos de Monte Carlo no sesgados: ejecutar un algoritmo *sesgado* 55%-correcto sólo 4 veces reduce la probabilidad de error a un 4,1%, puesto que $0{,}45^4 \approx 0.041$.

La ecuación 10.3 muestra que el número de repeticiones necesario para alcanzar un nivel de confianza dado (el 95% en este caso) depende fuertemente de la ventaja ε del algoritmo original. Una ventaja 10 veces menor requeriría aproximadamente 100 veces más repeticiones para obtener la misma fiabilidad. Por otra parte, no resulta mucho más oneroso obtener una confianza bastante mayor en la respuesta final. Por ejemplo, si hubiésemos deseado una probabilidad 10 veces más pequeña, habríamos utilizado

$$\Pr[X < E(X) - 2{,}576\sqrt{Var(\overline{X})}] \approx \tfrac{1}{2}\%$$

para concluir que basta con repetir un algoritmo de Monte Carlo no sesgado $(1/2 + \varepsilon)$-correcto un número de veces $6{,}636\left(\frac{1}{4\varepsilon^2} - 1\right)$ para hacer que sea 99,5%-correcto. Esto no es ni siquiera 2,5 veces más costoso que lograr una corrección del 95%. Existe un argumento de combinatoria que muestra que para llevar la probabilidad de error por debajo de δ para un algoritmo de Monte Carlo no sesgado cuya ventaja sea ε, el número de repeticiones necesario es proporcional a $1/\varepsilon^2$, tal como se vio, pero también a $\log \frac{1}{\delta}$. Véanse los problemas 10.24 y 10.25 para más detalles.

10.7 ALGORITMOS DE LAS VEGAS

Los algoritmos de Las Vegas toman decisiones probabilistas como ayuda para guiarse más rápidamente hacia una solución correcta. A diferencia de los algoritmos de Monte Carlo, nunca proporcionan una respuesta incorrecta. Hay dos categorías principales de algoritmos de Las Vegas. Pueden utilizar la aleatoriedad para guiar su búsqueda de tal manera que se garantice una solución correcta aun cuando se tomen decisiones poco afortunadas: si esto ocurriera, tan sólo necesitarán más tiempo. Alternativamente, pueden permitirse a sí mismos tomar caminos equivocados, que los llevarán a un callejón sin salida, haciendo imposible hallar una solución en esta ejecución del algoritmo.

Los algoritmos de Las Vegas del primer tipo suelen utilizarse cuando un algoritmo determinista conocido para resolver el problema en cuestión es mucho más rápido en el caso promedio que en el caso peor. *Quicksort*, de la Sección 7.4.2, ofrece el ejemplo más famoso; véase la Sección 10.7.2. Al incorporar un elemento aleatorio, se permite a los algoritmos de Las Vegas reducir, y a veces eliminar, esta diferencia entre casos buenos y malos. No es cuestión de evitar la aparición ocasional del comportamiento del algoritmo en el caso peor, sino más bien de romper la conexión entre la aparición de este comportamiento y el caso concreto que haya que resolver.

Recordemos de la Sección 2.4 que el análisis de la eficiencia en el caso del promedio de un algoritmo puede a veces producir resultados que induzcan a confusión. La razón es que todo análisis del caso medio debe estar basado en una hipótesis acerca de la distribución de probabilidad de los casos que haya que manejar. Una hipótesis que sea correcta para una aplicación dada del algoritmo puede resultar desastrosamente incorrecta para otra aplicación diferente. Supongamos, por ejemplo, que se utiliza *quicksort* como subalgoritmo dentro de un algoritmo más complejo. Vimos en la Sección 7.4.2 que requiere un tiempo que está en $\Theta(n \log n)$ para ordenar n elementos siempre y cuando los casos que haya que ordenar se seleccionen de forma aleatoria. Este análisis ya no tiene relación alguna con la realidad si de hecho entregamos al algoritmo casos que ya estén casi ordenados. En general, estos algoritmos deterministas son vulnerables a una distribución de probabilidad inesperada de los ca-

sos que algunas aplicaciones pueden pasarles para su resolución: incluso cuando las circunstancias del caso peor sean muy raras, podrían ser las más relevantes para la aplicación, dando lugar a una espectacular degradación del rendimiento. Los algoritmos de Las Vegas nos liberan de preocupaciones acerca de estas situaciones, igualando el tiempo necesario para diferentes casos de un tamaño dado.

El rendimiento de estos algoritmos de Las Vegas no es mejor que el del algoritmo determinista correspondiente cuando consideramos la media para todos los casos de un tamaño dado. Con mucha probabilidad, los casos que requirieran mucho tiempo con el método determinista se resuelven ahora mucho más deprisa, pero los casos en que el algoritmo determinista fuera especialmente bueno se ralentizan hasta un valor medio en el algoritmo de Las Vegas. De esta manera, estos algoritmos de Las Vegas «roban» tiempo de los casos «ricos» (los que resolvía rápidamente el algoritmo determinista) para dárselo a los casos «pobres». Llamamos a esto el *efecto Robin Hood*, y lo ilustramos en las Secciones 10.7.2 y 10.7.3. Esto es interesante cuando consideramos algoritmos deterministas que no son significativamente más rápidos que en el caso medio en el mejor de los casos, así que no se pierde gran cosa al reducirlos a la media, pero que sufren de unos pocos casos malos especialmente graves.

La otra categoría de algoritmos de Las Vegas se caracteriza por tomar de vez en cuando decisiones que los llevan a un callejón sin salida. Se pide que estos algoritmos sean capaces de reconocer su situación, en cuyo caso se limitarán a admitir el fallo. Este comportamiento sería intolerable en un algoritmo determinista, por cuanto supondría que no sería capaz de resolver el caso considerado. Sin embargo, la naturaleza probabilista de los algoritmos de Las Vegas hace que esta admisión de fallo sea aceptable siempre y cuando no se produzca con excesiva probabilidad: cuando se produzca un fracaso basta con volver a aplicar el mismo algoritmo al mismo caso para tener una nueva posibilidad de éxito. Hay problemas prácticos para los cuales esta disposición a arriesgarse a un fallo permite un algoritmo de Las Vegas eficiente, cuando no se conocen algoritmos deterministas que sean eficientes; véase la Sección 10.7.4. Aunque no se puede establecer una cota superior garantizada para el tiempo que se necesitará para obtener una respuesta si se reinicia el algoritmo siempre que falle, este tiempo puede ser razonable con una elevada probabilidad. Estos algoritmos no debieran de confundirse con aquellos que, como el algoritmo *simplex* para programación lineal, son extremadamente eficientes para la gran mayoría de los casos que hay que manejar, pero catastróficos para unos pocos casos: un algoritmo de Las Vegas deberá de tener un buen rendimiento esperado, sea cual sea el caso que haya que resolver.

Cuando se permite que falle un algoritmo de Las Vegas, es más cómodo representarlo en la forma de un **procedimiento** que en forma de una **función**. Esto permite tener un parámetro de salida *éxito*, que recibe el valor *verdadero* si se obtiene una solución, o *falso* en caso contrario. La llamada típica para resolver el caso x es $LV(x, y, éxito)$, en donde el parámetro de salida y recibe la

solución siempre que *éxito* reciba el valor *verdadero*. Por comodidad, escribiremos

devolver *éxito* ← *verdadero*

como equivalente de

éxito ← *verdadero*
devolver *éxito*

y el equivalente para **devolver** *éxito* ← *falso*.

Sea $p(x)$ la probabilidad de éxito del algoritmo cada vez que se le pide que resuelva el caso x. Para que un algoritmo merezca el nombre «Las Vegas», exigimos que $p(x) > 0$ para todo caso x. Esto asegura que se encontrará una solución eventualmente si seguimos repitiendo el algoritmo. Es aún mejor si existe una constante $\delta > 0$ tal que $p(x) \geq \delta$ para todo caso x, puesto que en caso contrario el número esperado de repeticiones antes del éxito podría crecer arbitrariamente con el tamaño del caso.

Consideremos el algoritmo siguiente:

función *RepetirLV*(x)
  repetir
  LV(x, y, *éxito*)
  hasta que *éxito*
  devolver y

Dado que cada llamada a $LV(x)$ tiene una probabilidad de éxito $p(x)$, el numero esperado de pasadas por el bucle es $1/p(x)$. Un parámetro más interesante es el tiempo esperado $t(x)$ antes de que *RepetirLV*(x) tenga éxito. Puede uno pensar a primera vista que $t(x)$ es simplemente $1/p(x)$ multiplicado por el tiempo esperado que requiera cada llamada a $LV(x)$. Sin embargo, un análisis correcto debe considerar por separado el tiempo esperado tomado por $LV(x)$ en caso de éxito y en caso de fracaso. Denotemos estos tiempos esperados mediante $s(x)$ y $f(x)$, respectivamente. Despreciando el tiempo que necesita el control del bucle **repetir**, $t(x)$ viene dado por un análisis de casos:

Con probabilidad $p(x)$, la primera llamada a $LV(x)$ tiene éxito al cabo de un tiempo esperado $s(x)$.
Con probabilidad $1 - p(x)$, la primera llamada a $LV(x)$ fracasa al cabo de un tiempo esperado $f(x)$. Después, volvemos al punto de partida, y seguimos estando a un tiempo esperado $t(x)$ del éxito. El tiempo esperado total en este caso es por tanto $f(x) + t(x)$.

Por tanto, $t(x)$ viene dado por una sencilla recurrencia

$$t(x) = p(x)s(x) + (1-p(x))(f(x) + t(x)),$$

que se resuelve fácilmente para dar

$$t(x) = s(x) + \frac{1 - p(x)}{p(x)} f(x) \tag{10.4}$$

Algunos algoritmos de Las Vegas permiten el ajuste fino de varios parámetros. Al girar un mando, por ejemplo, decrece tanto el tiempo esperado en el caso de éxito o de fracaso (lo cual es bueno) como la probabilidad de éxito (lo cual es malo). Cuando sucede esto, la ecuación 10.4 es la clave para optimizar el rendimiento global del algoritmo. Ilustramos esto en la Sección 10.7.1 donde volvemos a tratar al problema de las 8 reinas. La Sección 10.7.4 ofrece un ejemplo más útil y sofisticado de algoritmo de Las Vegas que puede fallar en ocasiones, pero para el cual hay una estrategia mejor que reiniciar todo el cálculo en caso de fracasar.

10.7.1 El problema de las ocho reinas, segunda parte

El problema de las ocho reinas proporciona un ejemplo instructivo de un algoritmo de Las Vegas que se beneficia de permitirle fallar. Recordemos que la técnica de retroceso empleada en la Sección 9.6.2 requiere explorar sistemáticamente los nodos del árbol implícito formado por los llamados vectores *k*-prometedores. Empleando esta técnica, obtenemos la primera solución después de examinar solamente 114 de los 2.057 nodos que hay en el árbol. Esto no está mal, pero el algoritmo no tiene en cuenta un hecho importante: en la mayoría de las soluciones, nada tienen de sistemático las posiciones de las reinas. Por el contrario, las reinas parecen haber sido situadas al azar.

Esta observación sugiere un algoritmo voraz de Las Vegas que vaya poniendo aleatoriamente reinas en filas sucesivas, ocupándose tan sólo de no poner nunca una nueva reina en una posición que sea amenazada por otra reina ya situada. Dado que éste es un algoritmo voraz, no se hace intento alguno de reubicar las reinas anteriores cuando ya no hay posibilidades para la reina que falta: estamos en un callejón sin salida. Como resultado, el algoritmo o bien acaba con éxito si se las arregla para situar todas las reinas en el tablero, o bien fracasa si no hay ningún cuadrado en el que se pueda añadir la reina siguiente. La aproximación voraz hace el algoritmo más sencillo de entender que el de vuelta atrás, y también más fácil de implementar a mano —aun cuando el código sea más largo— pero también implica una posibilidad de fracaso. ¿Se obtiene esta sencillez a costa de la eficiencia? Veremos que nada podría estar más lejos de la verdad. El algoritmo emplea los mismos conjuntos *col*, *diag*45 y *diag*135 que en la Sección 9.6.2 como ayuda para determinar qué posiciones siguen estando disponibles en la fila estudiada:

```
procedimiento reinasLV(var sol[1..8], éxito)
    vector ok[1..8] {contendrá las posiciones disponibles}
    col, diag45, diag135 ← Ø
    para k ← 0 hasta 7 hacer
        {sol[1..k] es k-prometedor; situemos la reina (k+1)-ésima}
        nb ← 0  {para contar el número de posibilidades}
        para j ← 1 hasta 8 hacer
            si j ∉ col y j-k ∉ diag45 y j+k ∉ diag135
            entonces {la columna j está disponible para la (k+1)-ésima reina}
                    nb → nb+1
                    ok[nb] ← j
        si nb = 0 entonces terminar éxito ← falso
        j ← ok[uniforme(1..nb)]
        col ← col ∪ {j}
        diag45 ← diag45 ∪ {j-k}
        diag135 ← diag135 ∪ {j+k}
        sol[k+1] ← j
        {fin del bucle para en k}
    devolver éxito ← verdadero
```

Para analizar este algoritmo, necesitamos determinar su probabilidad de éxito p, el número medio s de nodos que explora en caso de éxito, y el número medio f de nodos que explora en caso de fracaso. Claramente, $s = 9$, contando el vector vacío 0-prometedor y la solución 8-prometedora. Empleando una computadora, podemos calcular $p \approx 0{,}1293$ y $f \approx 6{,}971$. Por tanto, ¡se obtiene una solución más de una vez de entre ocho sin más que proceder de forma totalmente aleatoria! El número esperado de nodos explorados si repetimos el algoritmo hasta por alcanzar el éxito está dado por la ecuación 10.4: $s + \frac{1-p}{p} f \approx 55{,}93$, que es menos de la mitad del número de nodos explorados por vuelta atrás sistemática.

Podemos hacerlo aún mejor. El algoritmo de Las Vegas es demasiado derrotista: en cuanto encuentra un fallo, vuelve a empezar otra vez desde el principio. El algoritmo de vuelta atrás, por otra parte, hace una búsqueda sistemática de una solución que no tiene nada de sistemática, como sabemos. Una combinación juiciosa de ambos algoritmos sitúa primero aleatoriamente un cierto número de reinas en el tablero, y después utiliza la vuelta atrás para intentar añadir las reinas restantes, sin reconsiderar, sin embargo, las posiciones de las reinas que se hayan colocado aleatoriamente.

Una colocación aleatoria desafortunada de las posiciones de las primeras reinas puede hacer imposible añadir todas las demás. Esto sucede, por ejemplo, si las dos primeras reinas se ponen respectivamente en las posiciones 1 y 3. Cuantas mas reinas situemos aleatoriamente, menor será el tiempo medio que necesita la fase subsiguiente de vuelta atrás, tanto si tiene éxito como si falla, pero mayor será la probabilidad de fallo. Éste es el «mando de ajuste fino» que se mencionaba anteriormente. Hagamos que *detenerLV* denote el número de

reinas que se colocan aleatoriamente antes de pasar a la fase de vuelta atrás con $0 \leq detenerLV \leq 8$.

El algoritmo modificado es similar a *reinasLV* salvo que debemos incluir la declaración de un **procedimiento** *vueltaatrás* interno (véase más abajo), el bucle en k debe ir desde 0 hasta *detenerLV*–1, y sustituiremos la última línea (**terminar** *éxito* ← *verdadero*) por

vueltaatrás(*detenerLV*, *col*, *diag*45, *diag*135, *éxito*)

Esta instrucción invoca a la fase de vuelta atrás siempre y cuando el bucle no haya terminado prematuramente por un fallo. El **procedimiento** *vueltaatrás* se parece al algoritmo *reinas* de la Sección 9.6.2, salvo que tiene el parámetro adicional *éxito* y que termina inmediatamente después de hallar la primera solución, o después de determinar que no hay ninguna, según sea el caso.

Para dar al mando de ajuste fino su posición óptima, necesitamos determinar la probabilidad de éxito p, el número esperado s de nodos explorados en caso de éxito y el número esperado f de nodos explorados en caso de fracaso para cada valor posible de *detenerLV*. Entonces se puede utilizar la ecuación 10.4 para determinar el número esperado t de nodos explorados si se repite el algoritmo hasta que eventualmente encuentre una solución. Estos números, que se obtienen explorando todo el árbol de vuelta atrás con ayuda de una computadora, se resumen en la figura 10.3. El caso *detenerLV* = 0 corresponde a utilizar directamente el algoritmo determinista.

detenerLV	p	s	f	t
0	1,0000	114,00	—	114,00
1	1,0000	39,63	—	39,63
2	0,8750	22,53	39,67	28,20
3	0,4931	13,48	15,10	29,01
4	0,2618	10,31	8,79	35,10
5	0,1624	9,33	7,29	46,92
6	0,1357	9,05	6,98	53,50
7	0,1293	9,00	6,97	55,93
8	0,1293	9,00	6,97	55,93

Figura 10.3. Ajuste fino de un algoritmo de Las Vegas

Aunque una aproximación puramente probabilista (*detenerLV* = 8) es mejor que el determinismo puro (*detenerLV* = 0), una mezcla de ambas es mejor todavía. El compromiso más grato cuando se ejecuta a mano el algoritmo consiste en poner aleatoriamente las tres primeras reinas (*detenerLV* = 3) y seguir por vuelta atrás: esencialmente, esto falla en la mitad de las ocasiones, pero es muy rápido tanto si tiene éxito como si no. ¡Pruébelo, es divertido!

El número de nodos explorados es una buena medida de la cantidad de trabajo efectuado por el algoritmo, pero puede no dar una imagen completa de la situación.

Para asegurarnos, implementamos el algoritmo *reinasLV* en una estación de trabajo. La vuelta atrás pura encuentra la solución en 0,45 milisegundos, mientras que basta una media de 0,14 milisegundos si se sitúan dos reinas al azar antes de volver atrás. Esto es más de tres veces más rápido. Sin embargo, se necesitan 0,21 milisegundos de análisis en el caso medio si se sitúan al azar las tres primeras reinas, lo cual sólo es el doble de bueno que la vuelta atrás pura. Si se colocan todas las reinas de modo aleatorio, perdemos: se necesita análisis en el caso medio de casi un milisegundo para hallar una solución, que es casi el doble del tiempo necesario de forma determinista. Este resultado decepcionante se explica fácilmente: el tiempo necesario para generar valores pseudoaleatorios no se puede despreciar. Resulta que el 71% del tiempo invertido en resolver el problema de las ocho reinas cuando se sitúan aleatoriamente todas las reinas se emplea en generar valores pseudoaleatorios. Aun cuando nuestro rendimiento podría haber sido mejor si hubiéramos utilizado un generador más rápido —aunque menos sofisticado—¿debemos concluir que los beneficios de la aleatoriedad probablemente se compensan con coste adicional de la generación de pseudoaleatorios?

Una vez más, el problema de las ocho reinas se generaliza a un número de reinas n arbitrario. La ventaja ganada al utilizar la aproximación probabilística se hace más convincente a medida que crece n. Probamos el algoritmo en el problema de las 39 reinas. La vuelta atrás puramente determinista necesita examinar más de 10^{10} nodos antes de encontrar la primera solución: son 11.402.835.415 nodos para ser exactos. En comparación, el algoritmo de Las Vegas tiene éxito con una probabilidad aproximada del 21% si sitúa aleatoriamente las 29 primeras reinas, después de explorar solamente unos 100 nodos tanto si tiene éxito como si fracasa Con el análisis en el caso medio, se exploran menos de 500 nodos antes de alcanzar el éxito si se reinicia el algoritmo después de cada fallo. Esto es más de 20 millones de veces mejor que la vuelta atrás. Además, la mejora persiste cuando se considera el tiempo en una computadora real: se necesitan aproximadamente 47 horas de cálculo ininterrumpido en nuestra estación de trabajo para resolver el problema mediante vuelta atrás puramente determinista, mientras que el algoritmo de Las Vegas encuentra una solución cada 8,5 milisegundos. Si se emplea el algoritmo voraz de Las Vegas puro, la tasa de aciertos es aproximadamente de uno entre cada 135 intentos, pero cada intento es tan rápido que se encuentra una solución aproximadamente cada 150 milisegundos de análisis en el caso medio, lo cual sigue siendo casi un millón de veces más rápido que la vuelta atrás pura.

Si deseamos una solución del problema de las n reinas para un valor concreto de n, es evidentemente tonto analizar exhaustivamente todas las posibilidades para descubrir el valor óptimo de *detenerLV*, y aplicar entonces el algoritmo de Las Vegas en consecuencia. La evidencia empírica sugiere que el algoritmo de Las Vegas supera ampliamente la vuelta atrás pura siempre y cuando situemos casi todas —pero no todas— las reinas aleatoriamente. Por ejemplo, se minimiza el número esperado de nodos si colocamos aleatoriamente 88 reinas para el problema de las 100 reinas, y 983 para el problema de las 1.000 reinas. Aun cuando no se utilice el mejor valor posible para *detenerLV*, obtenemos nuestra

solución con razonable rapidez si no está demasiado alejado. Además, es fácil obtener muchas soluciones diferentes sin más que invocar repetidamente el algoritmo.

10.7.2 Selección y ordenación probabilistas

Volvemos al problema de hallar el k-ésimo menor elemento de un vector T de n elementos. Vimos en la Sección 7.5 un algoritmo que puede resolver este problema en un tiempo que está en $\Theta(n)$ en el caso peor, independientemente del valor de k. En particular, al seleccionar $k = \lfloor n/2 \rfloor$ se obtiene un algoritmo de tiempo lineal en el caso peor para calcular la mediana de un vector.

Recuerde que este algoritmo empieza por particionar los elementos del vector a ambos lados de un pivote, y que después, tal como en la búsqueda binaria, restringe su atención al subvector correspondiente. El proceso se repite hasta que son iguales todos los elementos que aún quedan por considerar, quizá porque sólo quede uno, en cuyo caso habremos hallado el valor deseado. Un principio fundamental de la técnica de divide y vencerás sugiere que cuanto más próximo a la mediana de los elementos esté el pivote, más eficiente será el algoritmo. A pesar de esto, no hay ni que pensar en seleccionar la mediana exacta como pivote: esto daría lugar a una recursividad infinita, porque hallar la mediana es un caso especial del problema de selección que estamos considerando. Por tanto, seleccionamos un pivote subóptimo conocido con el nombre de pseudomediana. Esto evita la recursividad infinita, pero sigue siendo relativamente costoso hallar la pseudomediana.

Por otra parte, también vimos una aproximación más sencilla que utiliza como pivote el primer elemento que quede por considerar. Esto nos asegura un tiempo de ejecución lineal de análisis en el caso promedio, pero con el riesgo de que el algoritmo requiera un tiempo cuadrático en el caso peor. A pesar de este caso peor prohibitivo, el algoritmo más sencillo tiene la ventaja de una constante oculta mucho más reducida, como consecuencia del tiempo que se ahorra al no calcular la pseudomediana. Cualquier estrategia determinista sencilla para seleccionar el pivote tiene bastantes probabilidades de dar lugar a un tiempo cuadrático en el caso peor para hallar la mediana, y a la inversa, los algoritmos lineales en el caso peor parecen necesitar una constante oculta muy grande. La decisión concerniente a si es más importante tener una ejecución eficiente en el caso peor o análisis en el caso medio debe de tomarse a la vista de la aplicación concreta. Si decidimos tener una mayor velocidad de análisis en el caso promedio mediante un algoritmo determinista más sencillo, debemos asegurarnos de que los casos que haya que resolver sean realmente seleccionados según la distribución uniforme. Una mala distribución de probabilidades de los casos podría significar el desastre.

Para que el tiempo de ejecución dependa solamente del número de elementos, pero no del ejemplar en sí, basta con seleccionar aleatoriamente el pivote entre los elementos que queden por considerar. El algoritmo resultante es muy similar a *selección* de la Sección 7.5:

```
función selecciónLV(T[1..n],s)
{Halla el s-ésimo elemento menor de T, 1 ≤ s ≤ n}
i ← 1;j ← n
repetir
    {La respuesta se encuentra en T[i..j]}
    p ← T[uniforme(i..j)]
    pivotebis(T[i..j], p, k, l)
    si s ≤ k entonces j ← k
    sino si s ≥ l entonces i ← l
        sino devolver p
```

El análisis solicitado en el problema 7.18 se aplica *mutatis mutandis* para concluir que el tiempo esperado que requiere este algoritmo probabilista de selección es lineal, independientemente del caso que haya que resolver. Por tanto, su eficiencia no se ve afectada por las peculiaridades de la aplicación en la cual se utilice el algoritmo. Siempre es posible que alguna ejecución particular del algoritmo requiera un tiempo cuadrático, pero la probabilidad de que suceda esto se vuelve cada vez más despreciable a medida que crece n y este suceso tan poco probable ya no está asociado a casos concretos.

Para resumir, hemos empezado con un algoritmo que es excelente cuando consideramos su tiempo medio de ejecución para todos los casos de algún tamaño dado, pero que es ineficiente en ciertos casos concretos. Empleando la aproximación probabilista, hemos transformado este algoritmo en un algoritmo de Las Vegas que es eficiente con una elevada probabilidad, sea cual fuere el caso considerado. De esta manera nos beneficiamos de los dos algoritmos deterministas vistos en la Sección 7.5: tenemos un tiempo esperado lineal en todos los casos, con una constante oculta reducida.

En cierta ocasión, indicamos a los alumnos de un curso de algoritmia que implementasen el algoritmo de selección que mejor les pareciera. Los únicos algoritmos vistos en clase eran los de la Sección 7.5. Dado que los alumnos no sabían qué ejemplares se iban a utilizar para probar sus programas —y temiendo lo peor por parte de sus profesores— ninguno se atrevía a correr el riesgo de emplear un algoritmo determinista con un caso peor cuadrático. Sin embargo, tres alumnos pensaron en emplear la aproximación probabilista. Esta idea les permitió derrotar a sus compañeros sin ningún esfuerzo: sus programas requerían una media de 300 milisegundos para resolver el caso trivial, mientras que la mayoría de los algoritmos deterministas requerían entre 1.500 y 2.600 milisegundos. Además, sus programas eran mucho más sencillos —y por tanto, menos proclives a contener errores sutiles— que los de sus compañeros.

Este mismo enfoque se puede emplear para transformar *quicksort* en un algoritmo que ordene n elementos en un tiempo esperado para el caso peor que está en $O(n \log n)$, mientras que el algoritmo que vimos en la Sección 7.4.2 requiere un tiempo que está en $\Omega(n^2)$ cuando el vector que hay que ordenar ya está ordenado. La versión aleatorizada de *quicksort* es como sigue. Para ordenar todo un vector T, basta llamar a *quicksortLV*(T[1..n]):

```
procedimiento quicksortLV(T[i, j])
    {Ordena el subvector T[i, j] en orden no decreciente}
    si j–i es suficientemente pequeño entonces insertar(T[i, j])
    sino
        p ← T[uniforme(i..j)]
        pivotebis(T[i.. j], p, k, l)
        quicksortLV(T[i..k])
        quicksortLV(T[l..j])
```

10.7.3 Tablas de dispersión universales

Recuerde de la Sección 5.6 que las tablas asociativas, como las que se utilizan para llevar la cuenta de los identificadores en un compilador, suelen implementarse por dispersión. Esto nos da un tiempo esperado constante de acceso a la tabla, siempre y cuando los símbolos de la tabla sean aleatorios. Desafortunadamente, esto no dice nada acerca del rendimiento de la dispersión en casos no aleatorios. Si por ejemplo, se emplea la dispersión para implementar la tabla de símbolos en un compilador, entonces la suposición de que todos los identificadores son igualmente probables no es razonable (¡afortunadamente!). Consiguientemente, ese análisis de caso medio puede inducir a confusión, y la probabilidad de los casos malos puede ser significativamente más alta que la esperada.

Hay algo más importante, y es que ciertos programas van a causar inevitablemente más colisiones que las esperadas sin que tenga la culpa el desafortunado programador. Estos programas se compilarán lentamente en todas las ocasiones, porque la función de dispersión se fija para siempre en el compilador. La sabiduría popular afirma que las cosas se igualan desde el punto de vista del sistema: algunos programas necesitarán más tiempo para compilarse que el esperado, pero otros irán más deprisa. Al final del día, la compilación habrá sido eficiente en el análisis del caso medio. Este punto de vista es inherentemente injusto. Si cada programa se compila muchas veces, siempre habrá unos pocos programas (y siempre serán los mismos) que requerirán un tiempo de compilación substancialmente mayor que el esperado. En un sentido real, estos programas pagan el coste para que todos los demás programas se compilen rápidamente. La dispersión de Las Vegas nos permite retener la eficiencia de la dispersión análisis en el caso medio, sin favorecer arbitrariamente a algunos programas a expensas de otros. Éste es el efecto Robin Hood en plena potencia: se le da a cada programa su justa proporción de los beneficios obtenidos por la dispersión. También, todos los programas pagarán de vez en cuando el precio de la eficiencia general, requiriendo un tiempo mayor que el esperado. Además, el buen rendimiento esperado de la dispersión de Las Vegas se puede demostrar matemáticamente sin hacer suposiciones acerca de la distribución de probabilidad de las secuencias de acceso a la tabla.

La idea básica de la dispersión de Las Vegas es que el compilador seleccione aleatoriamente la función de dispersión al principio de cada compilación, y que lo haga de nuevo cuando resulte necesaria una nueva dispersión. Esto asegura que

las listas de colisiones tienen una probabilidad alta de estar razonablemente bien equilibradas, sea cual fuere el conjunto de identificadores del programa que hay que compilar. Como resultado, un programa que dé lugar a un gran número de colisiones durante una compilación probablemente tendrá más suerte en la próxima ocasión en que se compile. Ahora bien, ¿qué quiere decir «seleccionar aleatoriamente la función de dispersión»?

La respuesta se encuentra en una técnica conocida con el nombre de *dispersión universal*. Sea U el universo de índices potenciales para la tabla asociativa, como puede ser el conjunto de todos los posibles identificadores si vamos a implementar un compilador, y sea $B = \{0, 1, 2, \ldots, N-1\}$ el conjunto de índices de la tabla de dispersión. Consideremos en U dos x e y diferentes. Supongamos primero que $h: U \to B$ es una función seleccionada aleatoriamente de entre todas las funciones que van de U a B según la distribución uniforme. Entonces la probabilidad $h(x) = h(y)$ es $1/N$. Esto se debe a que $h(y)$ podría tomar cualquiera de los N valores de B con igual probabilidad; en particular, el valor atribuido a $h(x)$ también se tomaría para $h(y)$ con probabilidad $1/N$. Sin embargo, U suele ser grande y hay demasiadas funciones de U en B para que sea razonable seleccionar aleatoriamente una de ellas según la distribución uniforme.

Considérese ahora un conjunto H de funciones de U en B, y considérense de nuevo en U dos x e y diferentes. Supongamos que $h: U \to B$ es una función seleccionada aleatoriamente de entre los miembros de H según la distribución uniforme. Decimos que H es una *clase universal$_2$ de funciones de dispersión* si la probabilidad de que sea $h(x) = h(y)$ es como máximo $1/N$. En otras palabras, requerimos que la probabilidad de que $h(x) = h(y)$ sea pequeña independientemente de los valores diferentes de x e y que se hayan considerado, siempre y cuando la selección de h se haga independientemente de estos valores. Vimos que el conjunto de todas las funciones de U en B es *universal$_2$*, pero demasiado grande para resultar útil. Las clases universales son interesantes porque pueden ser razonablemente pequeñas, de tal manera que en la práctica se puede seleccionar aleatoriamente una función de una de estas clases según la distribución uniforme. Además las funciones se pueden evaluar eficientemente. Daremos enseguida un ejemplo explícito de una de estas clases *universal$_2$* de funciones de dispersión, pero primero vamos a ver lo bien que funcionan para resolver el problema de la compilación.

Sea H una clase *universal$_2$* de funciones de dispersión de U en B. Sean x e y dos identificadores distintos cualesquiera. Por definición de universalidad, si seleccionamos aleatoriamente h dentro de H según la distribución uniforme, entonces la probabilidad de colisión entre x e y es como mucho de $1/N$. Considérese ahora un programa con n identificadores diferentes, y sea x uno cualquiera de ellos. Para cada uno de los $n - 1$ identificadores distintos de x, la probabilidad de colisión con x es como mucho $1/N$. Por tanto, el número esperado de identificadores en colisión con x es como mucho $(n - 1)/N$, que es menor que el factor de carga. Dado que esto es cierto para todo x, y puesto que mantenemos el factor de carga por debajo de uno por redispersión cuando es necesario, todo acceso a la tabla de dispersión requiere un tiempo esperado constante en el caso peor. Por tanto n accesos a la tabla de dispersión requieren un tiempo esperado en $\Theta(n)$ en el caso peor.

Compárese esto con el tiempo en $\Omega(n^2)$ que se requiere en el caso peor para la dispersión clásica. Tal como sucede con los algoritmos de Las Vegas, el «caso peor» hace alusión al peor conjunto posible de accesos a la tabla, y no a las peores selecciones aleatorias que pudiera hacer el compilador.

Se conocen varias clases universal$_2$ de funciones de dispersión. Daremos un ejemplo. Supongamos por sencillez que $U = \{0, 1, 2, \ldots, a-1\}$ y que $B = \{0, 1, 2, \ldots, N-1\}$. (Los identificadores deben transformarse en enteros empleando cualquier representación entera estándar de las cadenas de caracteres, como ASCII.) Sea p un número primo tan grande al menos como a. Sean i y j dos enteros. Definimos $h_{ij}: U \to B$ como

$$h_{ij}(x) = ((ix + j) \bmod p) \bmod N.$$

Entonces, $H = \{h_{ij} \mid 1 \le i < p \text{ y } 0 \le j < p\}$ es una clase *universal*$_2$ de funciones de dispersión de U en B. Seleccionar aleatoriamente una función en H es tan sencillo como seleccionar dos enteros menores que p. Además, se puede calcular eficientemente el valor de $h_{ij}(x)$, especialmente si hacemos que N sea una potencia de 2, lo cual simplifica la segunda operación del módulo.

10.7.4 Factorización de enteros muy grandes

Sea n un entero mayor que 1. El problema de *factorización* consiste en encontrar la descomposición única de n como producto de factores primos. El problema de *descomposición* consiste en hallar un divisor no trivial de n, siempre y cuando n sea compuesto. La factorización se reduce a la descomposición y a la comprobación de primalidad: para factorizar n, hemos acabado si n es primo; en caso contrario, se halla un divisor no trivial de m y se factorizan recursivamente m y n/m.

El algoritmo sencillo de descomposición es la prueba división, que halla el menor primo que sea divisor de n. Es inútil buscar divisores mayores que $\sqrt{n}$ porque si $m > \sqrt{n}$ divide a n, entonces también lo divide n/m, que es menor que $\sqrt{n}$:

```
función pruebadiv(n)
    para m ← 2 hasta [√n] hacer
        si m divide a n, entonces devolver m
    {Si el bucle no encuentra un divisor, n es primo}
    devolver n {un número primo es su menor divisor primo}
```

Este algoritmo requiere un tiempo que está en $\Omega(\sqrt{n})$ en el caso peor, que no tiene ninguna aplicación práctica ni siquiera para enteros de tamaño medio: si contamos tan sólo un nanosegundo para cada pasada por el bucle, se necesitarían miles de años para descomponer un número compuesto duro con unos cuarenta dígitos decimales, donde «duro» quiere decir que el número es el producto de dos primos de tamaños aproximadamente iguales.

El mayor número compuesto duro que se ha factorizado en el momento de escribir esto tiene 129 dígitos decimales. Esta factorización fue la clave para enfrentarse al desafío criptográfico RSA mencionado en la Sección 7.10. Recuerde

que se necesitaron ocho meses de cálculos en más de 600 computadoras de todo el mundo. Se ha estimado que habrían sido necesarios 5.000 años de cálculo ininterrumpido si se hubiese empleado una única estación de trabajo capaz de efectuar un millón de instrucciones por segundo. Aunque el esfuerzo es enorme, el éxito no habría sido posible sin un algoritmo sofisticado. De hecho, cuando se hizo el desafío en 1977, se estimó que la computadora más rápida disponible en aquel momento, ejecutando el mejor algoritmo conocido entonces, habría finalizado el cálculo ¡al cabo de dos millones de veces la edad del Universo! En esta sección damos tan sólo un esbozo de la idea básica que subyace al algoritmo victorioso.

Los algoritmos eficientes de descomposición se basan en el teorema siguiente, cuya sencilla demostración dejamos como ejercicio.

Teorema 10.7.1. *Sea n un número entero compuesto. Sean a y b dos enteros distintos entre* 1 *y* $n-1$ *tales que* $a + b \neq n$. Si $a^2 \bmod n = b^2 \bmod n$, *entonces* $mcd(a+b, n)$ *es un divisor no trivial de* n

Considérese por ejemplo $n = 2.537$. Sean $a = 2.012$ y $b = 1.127$. Obsérvese que $a^2 = 1595n + 1629$ y $b^2 = 500n + 1629$, tanto a^2 como b^2 son iguales a 1.629 módulo n. Dado que $a \neq b$ y que $a + b \neq n$, el teorema dice que $mcd(a+b, n) = mcd(1.319, 2.537) = 43$ es un divisor no trivial de n, lo cual es cierto. Esto sugiere un enfoque para descomponer n: buscar dos números distintos entre 1 y $n-1$ que tengan el mismo cuadrado módulo n, pero cuya suma no sea n, y utilizar el algoritmo de Euclides para calcular el máximo común divisor de su suma con n. Esto está muy bien siempre y cuando existan tales números cuando n es compuesto, y siempre y cuando podamos hallarlos eficientemente.

La primera cuestión se resuelve rápidamente. Supuesto que n tenga al menos dos divisores primos diferentes, $a^2 \bmod n$ admite al menos cuatro «raíces cuadradas» distintas en aritmética módulo n para cualquier a relativamente primo con respecto a n. Prosiguiendo con nuestro ejemplo, 1.629 admite exactamente cuatro raíces cuadradas módulo 2.537, a saber, 525, 1.127, 1.410 y 2.012. Estas raíces vienen en parejas: $525 + 2.012 = 1.127 + 1.410 = 2.537$. Dos cualquiera de estas raíces servirán, siempre y cuando no vengan de la misma pareja.

¿Y entonces, cómo podemos hallar a y b con la propiedad deseada? Éste es el momento en que entra en juego lo aleatorio. Sea k un entero que especificaremos después. Un entero se llama *k-uniforme*[3] si todos sus divisores primos se encuentran entre los k números primos más pequeños. Por ejemplo, $120 = 2^3 \times 3 \times 5$ es 3-suave, pero $35 = 5 \times 7$ no lo es. Cuando k es pequeño, los enteros k-suaves se pueden factorizar eficientemente mediante la prueba de la división. En su primera fase, el algoritmo de descomposición de Las Vegas selecciona aleatoriamente un entero x entre 1 y $n-1$, y calcula $y = x^2 \bmod n$. Si y es k-suave, entonces tanto x como la factorización de y se almacenan en una tabla. En caso contrario, seleccionamos aleatoriamente otro x. El proceso se repite hasta que se hayan encontrado $k+1$ en-

teros diferentes para los cuales se conozca la factorización de sus cuadrados módulo n.

Prosiguiendo con nuestro ejemplo en que $n = 2.537$, tomemos $k = 7$. Entonces nos conciernen solamente los primos 2, 3, 5, 7, 11, 13 y 17. Seleccionamos aleatoriamente un primer entero $x = 1.769$. Calculamos su cuadrado módulo n: $x^2 = 1.233n + 1.240$ y por tanto $y = 1.240$. No tiene éxito el intento de factorizar $1.240 = 2^3 \times 5 \times 31$, puesto que 31 no es divisible por ninguno de los primos admisibles. Un segundo intento con $x = 2.455$ es más afortunado: su cuadrado módulo n es $1.650 = 2 \times 3 \times 5^2 \times 11$. En este ejemplo pequeño, los intentos tienen éxito con una probabilidad aproximada del 20%. Prosiguiendo hasta conseguir 8 éxitos, obtenemos la tabla siguiente

$$
\begin{aligned}
x_1 &= 2.455 \quad y_1 = 1.650 = 2 \times 3 \times 5^2 \times 11 \\
x_2 &= 970 \quad y_2 = 2.210 = 2 \times 5 \times 13 \times 17 \\
x_3 &= 1.105 \quad y_3 = 728 = 2^3 \times 7 \times 13 \\
x_4 &= 1.458 \quad y_4 = 2.295 = 3^3 \times 5 \times 17 \\
x_5 &= 216 \quad y_5 = 990 = 2 \times 3^2 \times 5 \times 11 \\
x_6 &= 80 \quad y_6 = 1.326 = 2 \times 3 \times 13 \times 17 \\
x_7 &= 1.844 \quad y_7 = 756 = 2^2 \times 3^3 \times 7 \\
x_8 &= 433 \quad y_8 = 2.288 = 2^4 \times 11 \times 13
\end{aligned}
$$

Esta tabla se utiliza para formar una matriz M de dimensiones $(k+1) \times k$ sobre $\{0, 1\}$. Cada fila corresponde a un éxito; cada columna corresponde a uno de los primos admisibles. La entrada M_{ij} recibe como valor 0 si el j-ésimo primo aparece como potencia par (incluyendo el cero) en la factorización de y_i; en caso contrario $M_{ij} = 1$. Por ejemplo, $M_{1,1} = 1$ porque el primer primo, 2, aparece con elevado a 3 en y_3, y $M_{3,2} = 0$ porque el segundo primo, 3, aparece con la potencia par 0. Para seguir con nuestro ejemplo, obtenemos la matriz siguiente

$$
M = \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 0
\end{pmatrix}
$$

Dado que esta matriz contiene más filas que columnas, las filas no pueden ser linealmente independientes: debe existir un conjunto no vacío de filas cuya suma valga el vector cero en aritmética módulo 2. Este conjunto se puede hallar mediante la eliminación de Gauss-Jordan, aunque están disponibles métodos más eficientes cuando k es grande, sobre todo para matrices muy dispersas tales como las

que se obtienen mediante este algoritmo de factorización cuando n es grande. En nuestro ejemplo, hay siete soluciones distintas, tales como las filas 1, 2, 4 y 8, o las filas 1, 3, 4, 5, 6 y 7. Considere ahora lo que sucede si se multiplican las y_i correspondientes a las filas seleccionadas. Nuestros dos ejemplos producen

$$y_1 y_2 y_4 y_8 = 2^6 \times 3^4 \times 5^4 \times 11^2 \times 13^2 \times 17^2$$
$$y_1 y_3 y_4 y_5 y_6 y_7 = 2^8 \times 3^{10} \times 5^4 \times 7^2 \times 11^2 \times 13^2 \times 17^2$$

respectivamente. Los exponentes de estos productos son necesariamente pares por construcción. Por tanto, se puede obtener una raíz cuadrada de estos productos dividiendo por dos los exponentes.

En aritmética módulo n, se puede obtener una raíz cuadrada del mismo producto multiplicando los x_i correspondientes, puesto que cada $y_i = x_i^2$ mód n. En nuestro ejemplo, los dos enfoques para calcular una raíz cuadrada módulo n de $y_1 y_2 y_4 y_8$ dan lugar a

$$a = 2^3 \times 3^2 \times 5^2 \times 11 \times 13 \times 17 \text{ mód } 2.537 = 2.012$$
$$b = 2.455 \times 970 \times 1.458 \times 433 \text{ mód } 2.537 = 1.127.$$

Tal como vimos anteriormente, basta con calcular el máximo común divisor de $a + b$ y n para obtener un divisor no trivial de n. En general, esta técnica produce dos enteros a y b entre 1 y $n - 1$ tales que a^2 mod $n = b^2$ mod n. Sin embargo no hay garantía, de que $a \neq b$ ni de que $a+b \neq n$. De hecho, la utilización de $y_1 y_3 y_5 y_6 y_7$ en lugar de $y_1 y_2 y_4 y_8$ se obtiene

$$a' = 2^4 \times 3^5 \times 5^2 \times 7 \times 11 \times 13 \times 17 \qquad \text{mód. } 2.537 = 1.973$$
$$b' = 2.455 \times 1.105 \times 1.458 \times 216 \times 80 \times 1.844 \text{ mód. } 2.537 = 564$$

que resulta inútil porque $a' + b' = n$. Sin embargo, se puede demostrar que todo este proceso tiene éxito con una probabilidad mínima del 50% a no ser que mcd(a, n) sea un divisor no trivial de n, lo cual nos viene igualmente bien a efectos de la descomposición; véase el problema 10.41. Sin embargo al contrario del problema de las n reinas, no debemos volver a empezar partiendo de cero en caso de fallo. ¿Por qué vamos a descartar tanto trabajo bien hecho? En su lugar, buscaremos otros conjuntos de filas de M que sumen cero en aritmética módulo 2. Si esto tampoco funciona, buscaremos unos pocos pares más $\langle x_i, y_i \rangle$ y volveremos a probar con la matriz aumentada resultante.

Queda por determinar el valor de k que debe utilizarse para optimizar el rendimiento de esta aproximación. Cuanto mayor sea este parámetro, mayor será la probabilidad de que $x^2 \text{mod} n$ sea k-suave cuando seleccionemos x aleatoriamente. Por otra parte, cuanto más pequeño sea el parámetro, más deprisa podremos efectuar una prueba de k-suavidad, y más deprisa factorizaremos los valores k-suaves

que encontremos, y menos valores necesitaremos. La búsqueda de un compromiso óptimo requiere el uso de resultados avanzados de teoría de números. Sea

$$L = e^{\sqrt{\log n \log\log n}}$$

sea b un número real positivo arbitrario y sea $t = L^{1/2b}$. Se puede demostrar que si $k \approx L^b$, entonces aproximadamente un x de cada t es tal que x^2 mod n es k-suave. Dado que cada intento que no tiene éxito requiere k divisiones, y que se necesitan k+1 éxitos para construir la matriz, esta fase requiere un número esperado de divisiones de prueba que está aproximadamente en $O(tk^2) = O(L^{2b+1/2b})$, que se minimiza en $O(L^2)$ cuando $b =$. La búsqueda de un conjunto de filas que tengan como suma el vector cero requiere un tiempo que está en $O(k^3) = O(L^{3b})$ si se emplea la eliminación de Gauss-Jordan (una vez más, es posible mejorarlo), lo cual es despreciable en comparación con el tiempo que se necesita para construir la matriz si es $b =$. El cálculo final de un máximo común divisor por el algoritmo de Euclides es totalmente despreciable. De esta manera, si tomamos $k \approx \sqrt{L}$, el algoritmo produce la descomposición de n al cabo de un número esperado de divisiones que está en $O(L^2)$. Si n es un número medio de 100 dígitos decimales, entonces $L \approx 5 \times 10^{30}$ mientras que $\sqrt{n} \approx 7 \times 10^{49}$, que es 10^{19} veces mayor. Esto ilustra lo mucho mejor que es este algoritmo en comparación con las pruebas de división. Esta comparación no es enteramente imparcial, puesto que la constante oculta para la prueba de la división es más pequeña, pero incluso 10^{30} picosegundos son aproximadamente el doble de la edad estimada del Universo.

Hay varias mejoras que hacen más práctico el algoritmo. Si en lugar de seleccionar x aleatoriamente entre 1 y n–1 lo seleccionamos de tal manera que x^2 mod n tenga más probabilidades de ser k-uniforme, entonces reducimos el número de pruebas previas para obtener las k+1 relaciones necesarias. La aleatoriedad desempeña un papel fundamental en este algoritmo porque no hay ninguna aproximación determinista para hallar tantos x buenos que haya resultado ser eficiente. Sin embargo, hay heurísticas no demostradas que funcionan tan bien en la práctica que sería necio utilizar el algoritmo «puro» que se ha descrito más arriba. La más sencilla de éstas consiste en seleccionar los x ligeramente más grandes que $\sqrt{n}$; véase el Problema 10.42. Otra heurística, la *criba cuadrática*, funciona en un tiempo que está en $O(L)$. La factorización con éxito de un número *duro* de 129 dígitos decimales que se mencionaba al principio de esta sección se efectuó empleando un algoritmo algo más sofisticado: la «variante polinómica múltiple doble de la criba cuadrática para grandes primos». Otras técnicas de factorización interesantes son el método de la curva elíptica y la criba de cuerpos numéricos que exigen resultados relativamente avanzados de la teoría de números.

10.8 PROBLEMAS

Problema 10.1. Se tiene una moneda sesgada de tal forma que cada lanzamiento produce cara con una probabilidad p y *cruz* con una probabilidad complementaria $q = 1 - p$. Supongamos que cada lanzamiento de la moneda es independiente de los lanzamientos anteriores: la probabilidad de obtener *cara* en cualquier lanzamiento dado es exactamente p, independientemente de los resultados anteriores. Desafortunadamente, no se conoce el valor de p. Diseñar un proceso sencillo mediante el cual se pueda utilizar esta moneda para generar una secuencia de *bits* aleatorios perfectamente imparcial.

Problema 10.2. Decíamos en la Sección 10.5.1 que «basta» dejar caer alrededor de millón y medio de agujas para estimar π con una precisión de 0,01 en 95 ocasiones de cada 100. Esto se conseguía dejando caer agujas que tienen una longitud igual a la mitad de la anchura de las tablas que forman el suelo. Nuestra estimación de π era n/k, donde n es el número de agujas que se tiran y k es el número de agujas que caen sobre una rendija. Demostrar que se puede mejorar este «algoritmo» dejando caer agujas de longitud doble y produciendo $n/2k$ como estimación de π. ¿Cuántas de estas agujas es preciso dejar caer para tener al menos una probabilidad del 95% de obtener el valor correcto de π con una precisión de 0,01?

Problema 10.3. Otra aproximación probabilista para estimar el valor de π consiste en utilizar una integración de Monte Carlo para estimar el área de la cuarta parte de un círculo de radio 2. En otras palabras, podemos emplear la relación

$$\pi = \int_{x=0}^{2} \sqrt{4 - x^2}\, dx$$

¿Cuántos valores aleatorios de x es preciso utilizar para tener una probabilidad mínima del 95% de obtener el valor correcto de π con una precisión de 0,01?

Problema 10.4. Escribir un programa de computadora para simular el experimento de Buffon para estimar el valor de π. El desafío consiste en que no se permite utilizar el valor de π en el programa. Si no entiende por qué hay problemas, ¡intente hacerlo!

Problema 10.5. Considere la más sencilla de las estrategias probabilistas de conteo, en la cual el registro se incrementa con probabilidad 1/2 a cada llamada a *pulsar*, y *contar* devuelva el doble del valor almacenado en el registro; véase la Sección 10.5.3. Demostrar que el valor esperado proporcionado por esta estrategia es exactamente el número de llamadas a *pulsar*. ¿Cuál es la varianza del valor devuelto? Interprete esta varianza en términos de un intervalo de confianza.

Problema 10.6. En este problema analizaremos rigurosamente el algoritmo de conteo probabilista de la Sección 10.5.3. Demostrar que el valor esperado que devuelve una llamada a *contar* después de una llamada a *iniciar* seguida por m llamadas a *pulsar* es m, siempre y cuando ignoremos la improbable posibilidad de desbordamiento del registro. Para ello denotamos por $Pm(i)$ la probabilidad de que el registro contenga el valor i después de m llamadas a *pulsar*, en cuyo caso *contar* proporcionaría $2 - 1$. Está claro que $p_0(0) = 1$, $p_m(0) = 0$ para todo $m > 0$, y que $p_m(0) = 0$ para todo $i > m$. El registro contiene el valor i después de m pulsos bien si contenía el valor $i - 1$ después de $m - 1$ llamadas (con probabilidad $p_{m-1}(i - 1)$) y este valor incrementado en 1 con la siguiente llamada (con probabilidad $2^{-(i-1)}$), o bien si ya contenía el valor i después de m–1 pulsos (con probabilidad $p_{m-1}(i)$) y mantuvo su

valor en la llamada siguiente (con probabilidad $1 - 2^{-i}$). Por tanto:

$$p_m(i) = 2^{-(i-1)}p_{m-1}(i-1) + (1-2^{-i})p_{m-1}(i)$$

para todo $1 \le i \le m$. El valor esperado que devuelve *contar* al cabo de m llamadas a pulsar es

$$E(m) = \sum_{i=0}^{m} (2^i - 1)p_m(i)$$

Hay que demostrar por inducción matemática que $E(m) = m$ para todo $m \ge 0$.

Problema 10.7. Continuando con el problema 10.6, demostrar que la varianza del valor devuelto por una llamada a *contar* después de una llamada a *iniciar* seguida por m llamadas a pulsar es $m(m-1)/2$. Interpretar esto en términos de un intervalo de confianza.

Problema 10.8. Considere el algoritmo de conteo probabilista modificado que se especificaba en la Ecuación 10.1. Determinar la probabilidad con la cual *pulsar*(c) debería incrementar c. ¿Ha obtenido 2^{-c} cuando $\varepsilon = 1$, tal como debería? Rehaga el problema si se elimina la división por ε de la Ecuación 10.1. ¿Cuál sería el error garrafal en este caso?

Nota: En la práctica, las 2^n probabilidades relevantes se calcularían por anticipado y se mantendrían en una tabla, lo cual hace que la aproximación sólo resulte interesante si se necesita un gran número de registros, o si es aplicable el contexto del problema 10.10.

Problema 10.9. Prosiguiendo con el problema 10.8, ¿cuál es la varianza del valor devuelto por *contar* después de m llamadas a pulsar cuando se sigue la ecuación 10.1? Dé su respuesta como función de m y de ε. Interprétela en términos de un intervalo de confianza.

Problema 10.10. Las tarjetas inteligentes ofrecen una aplicación interesante para la técnica de conteo probabilista de la Sección 10.5.3. Las memorias de *sólo lectura* son técnicamente más fáciles de construir que las memorias ordinarias de lectura y escritura. Los *bits* de sólo lectura se inician a cero en la fábrica. Se pueden leer a voluntad, y se pueden poner a 1, pero no se pueden volver a poner a 0. Demuestre que es imposible contar más de n sucesos en un registro de n *bits* de sólo lectura empleando técnicas deterministas. Demostrar, sin embargo, que es posible contar hasta $2^n - 1$ sucesos por métodos probabilistas. En otras palabras, el conteo probabilista y un registro de sólo lectura tienen la misma aplicabilidad que el conteo determinista y un registro ordinario de igual longitud.

Problema 10.11. Una habitación contiene 25 desconocidos; ¿apostaría usted a la par que hay al menos dos que tengan el mismo cumpleaños?

Problema 10.12. Sea X un conjunto finito cuya cardinalidad n, deseamos conocer. Desafortunadamente, n es demasiado grande para que resulte práctico limitarnos a contar los elementos uno por uno. Supongamos, por otra parte, que somos capaces de seleccionar aleatoriamente elementos de X según la distribución uniforme, mediante una llamada a *uniforme*(X). Considere el algoritmo siguiente:

```
función card(X)
    k ← 0
    S ← Ø
    a ← uniforme(X)
    repetir
        k ← k+1
        S ← S ∪ {a}
        a ← uniforme(X)
    hasta que a ∈ S
    devolver k²/2
```

Demostrar que este algoritmo devuelve una estimación no sesgada del número n de ele-

mentos de X y que se ejecuta en un tiempo esperado que está en $\Theta(\sqrt{n})$ si se pueden efectuar las llamadas a *uniforme*(X) y las operaciones sobre el conjunto S con un coste unitario. Si no puede demostrar esto rigurosamente (¡es bastante difícil!), dé un argumento convincente por el cual sea razonable creer que el número de elementos del conjunto es aproximadamente el cuadrado del número de extracciones independientes de X antes de que se produzca la primera repetición. Quizá le sirva de ayuda resolver primero los problemas 5.14 y 10.11.

Problema 10.13. El algoritmo probabilista de conteo del problema 10.12 es eficiente en términos de tiempo, pero puede no ser práctico en términos de espacio de almacenamiento como consecuencia de la necesidad de llevar la cuenta del conjunto S. Utilice de la mejor manera posible la generación pseudoaleatoria para modificar el algoritmo, de tal manera que requiera una cantidad de espacio constante sin incrementar su tiempo de ejecución, salvo por un pequeño factor constante. Éste es uno de los raros casos en los que utilizar un generador verdaderamente aleatorio sería más estorbo que ventaja aun cuando se paga un precio: ya no se puede demostrar matemáticamente la corrección del algoritmo modificado.

Problema 10.14. Buscar un algoritmo de Monte Carlo eficiente para decidir, dadas dos matrices $n \times n$ A y B, si B es o no la inversa de A. En términos de n, y de la probabilidad aceptable de error ε, ¿cuánto tiempo requiere el algoritmo?

Problema 10.15. Mostrar que los testigos falsos fuertes de primalidad son automáticamente testigos falsos con respecto a la comprobación de Fermat; véase la Sección 10.6.2.

Sugerencia. Utilizar el hecho de que $(n-1)^2 \bmod n = 1$.

Problema 10.16. Demostrar el Teorema 10.6.2.

Problema 10.17. El algoritmo *primoaleatorio* de la Sección 10.6.3 genera aleatoriamente números probablemente primos mediante selección aleatoria repetida de enteros impares, hasta que se encuentra uno que pasa por un número suficiente de aplicaciones de la comprobación de Miller-Rabin. Explicar como difiere el resultado si en lugar de hacer esto seleccionamos un punto de partida impar aleatorio y lo incrementamos sucesivamente de dos en dos hasta obtener un número que pase el mismo número de aplicaciones de la misma comprobación.

Problema 10.18. Vimos que 561 es el caso peor para la comprobación de *primalidad* de Fermat, entre todos los compuestos impares menores que 1.000. Esto es cierto siempre y cuando consideremos la probabilidad de error de la comprobación. Sin embargo, existe un número compuesto menor que 1.000 y que admite incluso más testigos falsos que 561 ¿Cuál es? ¿Cuántos de estos testigos falsos son también testigos falsos fuertes?

Problema 10.19. El *teorema del número primo* afirma que el número de números primos menores que n es aproximadamente $n / \log n$. (Recuerde que «log» denota el logaritmo natural.) Esta aproximación es bastante precisa. Por ejemplo, existen 50.847.478 primos menores que 10^9, mientras que $n / \log n \approx 48.254.942$ cuando $n = 10^9$. Estimar la probabilidad de que un número entero impar de 100 cifras seleccionado aleatoriamente según una distribución uniforme sea primo.

Sugerencia: El número de primos de 100 cifras es igual al número de primos menores que 10^{100} menos el número de primos menores que 10^{99}.

Problema 10.20. Considere el siguiente programa que no termina:

```
programa imprimeprimos
imprimir 2, 3
n ← 5
repetir
    si RepetirMillRab(n, ⌊lg n⌋)
    entonces imprimir n
    n ← n+2
adnauseam
```

Claramente, este programa imprime eventualmente todos los números primos. Uno podría esperar también que se produzcan erróneamente números compuestos de vez en cuando. Demostrar que es improbable que suceda esto. Más precisamente, demostrar que hay una probabilidad mayor que 99% de que *nunca* se produzca un número compuesto mayor que 100, independientemente del tiempo que permitamos que corra el programa.

Pista: Esta cifra del 99% es muy prudente, por cuanto seguiría siendo válida aun cuando *MillRab*(n) tuviese una tasa exacta del 25% de fallos en todos los números compuestos.

Problema 10.21. En la Sección 10.6.2 se veía un algoritmo de Monte Carlo capaz de decidir la *primalidad* que siempre era correcto cuando se le daba un número primo, y que era correcto con una probabilidad mínima de 3/4 cuando se le daba un número compuesto. El tiempo de ejecución del algoritmo para una entraba n estaba en $O(\log^3 n)$. Buscar un algoritmo de Monte Carlo que siempre sea correcto cuando se le de un número compuesto, y que sea correcto con una probabilidad mínima de 1/2 cuando se le dé un número primo. El tiempo de ejecución del algoritmo debe estar en un tiempo que esté en $O(\log^k n)$ para alguna constante k.

Problema 10.22. En la Sección 10.6.4 estudiamos la amplificación de la ventaja estocástica de algoritmos de Monte Carlo no sesgados para problemas de decisión. Aquí investigamos la situación para problemas que puedan tener más de dos respuestas posibles. Los casos de problemas generales pueden tener más de una respuesta *correcta*, como. Piense, por ejemplo, en el problema de las ocho reinas, o en el problema de buscar divisores no triviales de un número entero compuesto. Cuando se resuelven problemas como estos mediante algoritmos probabilistas, puede suceder que se obtengan distintas respuestas correctas cuando se ejecute un mismo algoritmo varias veces con la misma entrada. Vimos en la Sección 10.7 que ésta es una virtud de los algoritmos de Las Vegas, pero puede ser algo catastrófico en lo concerniente a algoritmos de Monte Carlo no sesgados.

Recuerde que un algoritmo de Monte Carlo es p-correcto cuando devuelva una respuesta correcta con una probabilidad mínima p, sea cual fuere el caso considerado. La posible dificultad es que aun cuando un algoritmo p-correcto devuelve una respuesta correcta con elevada probabilidad cuando p es grande, podría ocurrir que una respuesta equivocada sistemática apareciera más frecuentemente que cualquier respuesta correcta. En este caso, la amplificación de la ventaja estocástica por votación de la mayoría ¡reduciría la probabilidad de estar en lo cierto! Demostrar que si el algoritmo *MC* es 75%-correcto, puede ocurrir que *MC*3 no sea ni siquiera 71%-correcto, donde *MC*3 devuelve la respuesta más frecuente entre tres llamadas a *MC*, como en la Sección 10.6.4. (Se deshacen arbitrariamente los empates.) ¿Para qué valor de k podría ser *RepetirMC*(·, k) menos que 50%-correcto aunque *MC* fuera 75%-correcto?

Problema 10.23. Sea *MC* un algoritmo de Monte Carlo p-correcto no sesgado y consideremos el algoritmo *RepetirMC*(·, k) de la Sec-

ción 10.6.4, que ejecuta k veces el algoritmo MC y devuelve la respuesta más frecuente. Si k es par se produce un problema en el caso de un empate. El código de *RepetirMC*(·, k) en la Sección 10.6.4 devuelve el valor *falso* en este caso (puesto que $V = F$). Esto degrada la probabilidad de un resultado correcto para casos en los que la respuesta correcta sea *verdadero*. Una solución mejor sería lanzar una moneda imparcial en caso de empate, para decidir la respuesta que hay que devolver. Demuestre que si se modifica *RepetirMC* con estas ideas, la probabilidad de que *RepetirMC*(·, k) devuelva la respuesta correcta cuando k es par resulta ser exactamente igual a la probabilidad de que *RepetirMC*(·, $k - 1$) devuelva la respuesta correcta. Concluya que nunca es una buena idea repetir un algoritmo de Monte Carlo no sesgado un número par de ocasiones para amplificar la ventaja estocástica.

Problema 10.24. Sean ε y δ dos números reales positivos tales que $\varepsilon + \delta < 1/2$. Sea MC un algoritmo de Monte Carlo $(1/2 + \varepsilon)$-correcto no sesgado para un problema de decisión. Empleando únicamente argumentos combinatorios elementales, demostrar que *RepetirMC*(·, k) es $(1-\delta$-correcto siempre y cuando $k \geq \frac{1}{2\varepsilon^2} \log 1/\delta$. En otras palabras, basta repetir un algoritmo de Monte Carlo cuya ventaja sea ε este número k de veces para obtener un algoritmo de Monte Carlo cuya probabilidad de error sea como mucho δ. (Recuerde que «log» denota el logaritmo natural.) Esta fórmula es excesivamente prudente. Sugiere repetir unas 600 veces un algoritmo de Monte Carlo 55%-correcto no sesgado para lograr una corrección del 95%, mientras que en la Sección 10.6.4 vimos que bastaban 269 repeticiones.

Sugerencia: Utilice la ecuación 10.2 y el hecho consistente en que $-2/\lg(1 - 4\varepsilon^2) < (\log 2)/2\varepsilon^2$.

Problema 10.25. Prosiguiendo el problema 10.24, demostrar que si un algoritmo de Monte Carlo no sesgado cuya ventaja sea ε se repite $k = 2m - 1$ veces, y si nos quedamos con la respuesta más frecuente, el algoritmo resultante es $(1-\delta$ correcto, en donde

$$\delta = \frac{1}{2} - \varepsilon \sum_{i=0}^{m-1} \binom{2i}{i} \left(\frac{1}{4} - \varepsilon^2\right)^i \leq \frac{\left(1 - 4\varepsilon^2\right)^m}{4\varepsilon\sqrt{\pi m}} \qquad (10.5)$$

La primera parte de esta fórmula es útil para calcular la probabilidad exacta de error que resulta de la amplificación de la ventaja estocástica. Se puede obtener rápidamente una buena cota superior del número de repeticiones necesarias para pasar de la ventaja ε a la probabilidad de error δ a partir de la segunda parte. Por ejemplo, nos dice que es suficiente repetir 303 veces un algoritmo 55%-correcto para lograr una corrección del 95%. Esto es mejor que las 600 veces que sugería la fórmula del Problema 10.24, pero no llega a ser 269, que obtenemos a partir del Teorema Central del Límite empleando las tablas de distribución normal. Sin embargo, este método tiene la ventaja de que no requiere que estén disponibles estas tablas, y sigue siendo válido incluso para valores pequeños de k, cuando el Teorema Central del Límite no es adecuado.

Problema 10.26. Prosiguiendo con el problema 7.36, dar un algoritmo de Monte Carlo sesgado y 1/2-correcto para decidir si un vector T contiene o no un elemento mayoritario. El algoritmo debe ejecutarse en un tiempo lineal, y las únicas comparaciones que se admiten entre los elementos de T son comprobaciones de igualdad. Observe que el único mérito de este algoritmo es su sencillez, puesto que el algoritmo determinista solicitado en el problema 7.36 resuelve el problema en tiempo lineal con una constante oculta muy pequeña.

Problema 10.27. Demostrar que el problema de primalidad se puede resolver mediante un algoritmo de Las Vegas cuyo tiempo de ejecución esperado está en $O(\log^k n)$ para alguna constante k. Puede tomarse como base el algoritmo de Monte Carlo que se necesita para el problema 10.21.

Problema 10.28. En el espíritu del problema 10.27, sean A y B dos algoritmos de Monte Carlo sesgados para resolver el mismo problema de decisión. El algoritmo A es p-correcto pero su respuesta está garantizada cuando es *verdadero*; el algoritmo B es q-correcto pero su respuesta está garantizada cuando es *falso*. Mostrar la forma de combinar A y B en un algoritmo $LV(x, y, éxito)$ para resolver el mismo problema. Una llamada a LV no debe de requerir un tiempo significativamente mayor que una llamada a A seguida por una llamada a B. Si el algoritmo de Las Vegas tiene éxito con una probabilidad mínima r sea cual fuere el caso, ¿cuál es el mejor valor de r que podemos obtener?

Problema 10.29. Sea X un conjunto finito cuyos elemento son fáciles de enumerar y sea Y un subconjunto no vacío de X, de cardinalidad desconocida. Suponga que puede decidir, dado un $x \in X$, si x pertenece o no a Y. ¿Cómo seleccionaría un elemento aleatorio de Y según la distribución uniforme? La respuesta evidente es hacer una primera pasada sobre X para contar el número de n de elementos que hay en Y, seleccionar entonces un número entero aleatorio $k \leftarrow$ *uniforme*$(1..n)$, y localizar finalmente el k-ésimo elemento de Y recorriendo de nuevo X, a no ser que se guarden en un vector los elementos de Y durante la primera pasada sobre X. Sorprendentemente, se puede resolver este problema con una sola pasada sobre X, sin espacio adicional, y sin contar primero los elementos de Y. Considere el algoritmo siguiente.

```
función dibujar(X, Y)
    n ← 0
    para cada x ∈ X hacer
        si x ∈ Y entonces n ← n + 1
                          si uniforme(1..n) = n
                          entonces z ← x
    si n > 0 entonces devolver z
             sino devolver "¡Error! ¡Y está
                            vacío!"
```

Demostrar por inducción matemática sobre el número de elementos de Y que este algoritmo halla aleatoriamente un elemento de Y según la distribución uniforme. Modificar el algoritmo *reinasLV* de la Sección 10.7.1 para incorporar esta técnica, seleccionando aleatoriamente según la distribución uniforme entre las posiciones que sigan abiertas para la reina siguiente, aunque no sabemos *a priori* cuántas hay. Esto hace el algoritmo más elegante pero menos eficiente, porque requiere más llamadas al generador pseudoaleatorio.

Problema 10.30. Ejecutar manualmente el algoritmo de Las Vegas para el problema de las ocho reinas. Coloque aleatoriamente las tres primeras reinas en las tres primeras filas e intente completar la solución por vuelta atrás. Si fracasa, vuelva a empezar. Como dijimos, la probabilidad de éxito es aproximadamente del 50% en cada intento. Resuelva también el problema a mano con vuelta atras pura. ¿Que método conduce antes a una solución?

Problema 10.31. Implemente en una computadora el algoritmo de Las Vegas para el problema de las n reinas que sitúa aleatoriamente *detenerLV* primeras reinas antes de intentar completar por vuelta atrás la solución parcial. Experimentar para resolver el problema de las 39 reinas con distintos valores de *detenerLV*. Hallar una solución para el

problema de las 100 reinas, y otra para el problema de las 1.000 reinas.

Problema 10.32. Demostrar que si se implementa la tabla de símbolos de un compilador mediante dispersión universal, y se mantiene por debajo de 1 el factor de carga, la probabilidad de que cualquier identificador esté en colisión con más de otros t es menor que $1/t$, para cualquier entero t. Concluir que el tiempo medio necesario para una sucesión de n accesos a la tabla está en $O(n)$. Compare este resultado con el del Problema 5.16.

Problema 10.33. Sea U un conjunto con a elementos. ¿Cuántas funciones de U en {0, 1, 2,..., N–1} existen? ¿Cuántos *bits* se necesitan análisis en el caso medio para escribir una descripción de una de estas funciones?

Problema 10.34. Demostrar que la clase de funciones de dispersión dada al final de la Sección 10.7.3 es universal$_2$.

Problema 10.35. Buscar aplicaciones de la dispersión universal que no tengan nada que ver con la compilación, ni siquiera con la implementación de una tabla asociativa.

Problema 10.36. Normalmente, los elementos de una lista se dispersan por toda la memoria de la computadora. Una lista *compacta* de longitud n se implementa mediante dos vectores $val[1..n]$ y $ptro[1..n]$, y un entero cabeza. El primer elemento de la lista se encuentra en $val[cabeza]$, el siguiente se encuentra en $val[ptro[cabeza]]$ y así sucesivamente. En general, si $val[i]$ no es el último elemento de la lista, $ptro[i]$ da el índice del siguiente elemento en val. El final de la lista se marca mediante $ptro[i] = 0$. Considere ahora una lista compacta cuyos elementos están en orden no decreciente. Sea x un elemento. El problema es localizar x dentro de la lista. La búsqueda binaria no es posible porque no hay una forma directa de buscar el punto medio de una lista, tanto si es compacta como si no.

(a) Demostrar que todo algoritmo determinista para este problema requiere un tiempo que está en $\Omega(n)$ en el caso peor.

(b) Diseñar un algoritmo de Las Vegas capaz de resolver este problema en un tiempo esperado en $O(\sqrt{n})$ en el caso peor.

Sugerencia: Examine $\sqrt{n}$ puntos seleccionados aleatoriamente en la lista, y comience la búsqueda en el mayor de estos puntos que no sea mayor que el objetivo, x. ¿Qué hay que hacer si todos los puntos son mayores que el objetivo?

Problema 10.37. Demostrar el teorema 10.7.1.

Problema 10.38. Buscar un x distinto de los dados en la Sección 10.7.4 tal que $1.000 \leq x \leq 2.000$ y $x^2 \bmod 2573$ es 7-uniforme.

Problema 10.39. En la Sección 10.7.4 veíamos dos soluciones para el problema de hallar un conjunto no vacío de filas de la matriz M que tengan como suma el vector cero en aritmética módulo 2. Buscar las otras cinco soluciones. Para cada una, determinar si lleva a un divisor no trivial de $n = 2.537$.

Problema 10.40. Sea n un número compuesto que tiene al menos dos divisores primos distintos, y sea a relativamente primo con n. Demostrar que $a^2 \bmod n$ admite al menos cuatro raíces cuadradas distintas en aritmética módulo n.

Problema 10.41. Sea n un número compuesto que tiene al menos dos divisores primos distintos, y sea $\langle a,b \rangle$ la primera pareja que se obtiene mediante el algoritmo de des-

composición de Las Vegas de la Sección 10.7.4. Demostrar que tanto mcd(a, n) como mcd(a+b, n) son divisores no triviales de n con una probabilidad mínima del 50%.

Sugerencia: Si mcd(a, n) = 1 entonces mcd(x_i, n) = 1 para todo x_i que formase parte de b. Se toma uno de estos x_i arbitrariamente. Sabemos por el problema 10.40 que $y_i = x_i^2 \bmod n$ tiene al menos cuatro raíces cuadradas distintas módulo n, incluyendo a x_i. Demostrar que si se hubiera seleccionado aleatoriamente cualquier otra raíz distinta de x_i y de $n - x_i$, en vez de entonces x_i la descomposición habría tenido éxito. Haga la conclusión necesaria.

Problema 10.42. Al final de la Sección 10.7.4 afirmábamos que se esperaba que la probabilidad de que x^2 mód n fuera k-uniforme mejorase si se tomaba un x ligeramente mayor que $\sqrt{n}$, en lugar de seleccionarlo aleatoriamente entre 1 y $n - 1$. Dar una razón intuitiva convincente en apoyo de esta afirmación.

Sugerencia: Demostrar que la longitud binaria de $\lceil\sqrt{n}\,\rceil^2 \bmod n$ es como máximo aproximadamente la mitad de un cuadrado aleatorio módulo n. ¿Qué sucede con la longitud de $(\lceil\sqrt{n}\,\rceil + i)^2 \bmod n$ para valores pequeños de i?

10.9 REFERENCIAS Y TEXTOS MÁS AVANZADOS

Los primeros ejemplos históricos de algoritmos probabilistas se retrotraen hasta las culturas «primitivas» en *Shallit* (1992). El término «Monte Carlo», que fue introducido en la literatura por *Metropolis* y *Ulam* (1949) ya se empleaba en el secreto mundo de la investigación atómica durante la Segunda Guerra Mundial, y en particular en Los Álamos, Nuevo Méjico. Recuerde que «Monte Carlo» suele utilizarse para describir cualquier algoritmo probabilista, en contra de la acepción empleada en este libro. El término «Las Vegas» fue introducido por *Babai* (1979) para distinguir los algoritmos probabilistas que responden correctamente (si es que responden) de aquellos que de vez en cuando cometen un error.

Dos fuentes enciclopédicas de técnicas para generar números pseudoaleatorios son *Knuth* (1969) y *Devroye* (1986). El primero de ellos incluye pruebas para intentar distinguir una secuencia pseudoaleatoria de una que sea realmente aleatoria. La solución del problema 10.1 es de *Von Neumann* (1951). Nosotros hemos utilizado el muy recomendable generador pseudoaleatorio dado por *L'Écuyer* (1988, 1990) en nuestros experimentos con el problema de las n reinas. Un generador más interesante desde el punto de vista criptográfico se da en *Blum* y *Micali* (1984); este artículo y el de *Yao* (1982) presentan la noción de un *generador impredecible*, que puede pasar cualquier prueba estadística que se pueda efectuar en un tiempo polinómico. El generador descrito al final de la Sección 10.4 es de *Blum, Blum* y *Shub* (1986). Se pueden hallar más referencias acerca de este tema en *Brassard* (1988). En Vazirani (1986, 1987), se dan técnicas generales para enfrentarse con generadores que estén parcialmente en manos del adversario.

El experimento diseñado por *Georges Louis Leclerc* (1777), conde de Buffon, fue llevado a cabo varias veces en el siglo XIX; véase por ejemplo *Hall* (1873). El proceso mediante el cual se puede utiliza para estimar π se analiza con detalle en *Sólomon* (1978). Un texto estándar de estadística matemática y análisis de datos es el de *Rice* (1988). Para un texto anterior de algoritmos numéricos probabilistas, consulte *Sobol'* (1974). En *Fox* (1986) se pone de manifiesto que los métodos puros de Monte Carlo no son especialmente buenos para la integración numérica con una *dimensión*

fija: es preferible seleccionar los puntos sistemáticamente para que estén bien espaciados, técnica que se conoce con el nombre de «cuasi Monte Carlo». El conteo probabilistas proviene de *Morris* (1978); véase *Flajolet* (1985) para un análisis detallado. Se da una solución del problema 10.12 en *Brassard* y *Bratley* (1988) pero tenga en cuenta que era incorrecta en las dos primeras impresiones: el análisis correcto les fue proporcionado a los autores por *Philippe Flajolet*. Para una aplicación criptográfica, véase *Kaliski, Rivest* y *Sherman* (1988). Se discute otro estilo de conteo probabilista en *Flajolet* y *Martin* (1985). Los algoritmos numéricos probabilistas diseñados para resolver problemas de álgebra lineal se discuten en *Curtiss* (1956), *Vickery* (1956), *Hammersley* y *Handscomb* (1965) y *Carasso* (1971). En *Bratley, Fox* y *Schrage* (1983) se encontrará una guía de la Simulación.

El algoritmo de Monte Carlo para verificar la multiplicación matricial procede de *Freivalds* (1979); véase también *Freivalds* (1977). La comprobación de primalidad de Monte Carlo que presentamos aquí es equivalente a la que hay en *Rabin* (1976, 1980 b); se basa en un trabajo anterior de *Miller* (1976). Otra comprobación de primalidad por el método de Monte Carlo fue descubierta independientemente por *Sólovay* y *Strassen* (1977). El número esperado de testigos falsos de primalidad para un entero compuesto aleatorio se investiga en *Erdős* y *Pomerance* (1986); véase también *Monier* (1980). El hecho de que baste probar la pseudoprimalidad fuerte con las bases 2, 3, 5, 7 y 61 para decidir de forma determinista si un numero menor que 10^{13} es primo fue descubierto por *Claude Goutier*. La demostración de que la comprobación de *Fermat* puede ser incorrecta arbitrariamente se sigue de *Alford, Granville* y *Pomerance* (1994). La discusión acerca de la generación de primos aleatorios procede de *Beauchemin, Brassard, Crépeau, Goutier* y *Pomerance* (1988); véase también *Kim* y *Pomerance* (1989) y *Damgärd, Landrock* y *Pomerance* (1993). En *Couvreur* y *Quisquater* (1982) y en *Maurer* (1995) se dan métodos eficientes para generar primos aleatorios certificados. Se da una solución teórica del Problema 10.21 en *Goldwasser* y *Kilian* (1986) y en *Adleman* y *Huang* (1992). Para más información sobre comprobaciones de primalidad y de su implementación, consulte *Williams* (1978), *Lenstra* (1982), *Adleman, Pomerance* y *Rumely* (1983), *Kranakis* (1986), *Cohen* y *Lenstra* (1987), *Koblitz* (1987) y *Bressoud* (1989). En el clásico de *Hardy* y *Wright* (1938) se encontrará más información acerca de la Teoría General de Números.

La aproximación de Las Vegas al problema de las ocho reinas les fue sugerida a los autores por *Manuel Blum*. *Pageau* (1993) desarrolló otras investigaciones posteriores. Para más información general acerca del problema, consulte las referencias dadas en la Sección 9.10. El término «Robin Hood» apareció en *Celis, Larson* y *Munro* (1985) en un contexto determinista. Se atribuye a *Floyd* (1978) un primer algoritmo probabilista para la búsqueda de la mediana que tiene un tiempo esperado lineal: véase el ejercicio 5.3.3.13 de *Knuth* (1973). Es anterior al algoritmo determinista clásico de tiempo lineal en el caso peor descrito en la Sección 7.5. En *Rivest* y *Floyd* (1973) se da un algoritmo probabilista capaz de hallar el i-ésimo menor elemento de entre n elementos en un número esperado de comparaciones que está en $n + i + O(\sqrt{n})$. La dispersión universal fue inventada por *Carter* y *Wegman* (1979); véase también *Wegman* y *Carter* (1981). Un primer algoritmo de factorización de enteros debido a *Pollard* (1975) tiene cierto aroma probabilista. El algoritmo probabilista de factorización de enteros que se describe aquí es de *Dixon* (1981) pero está basado en ideas propuestas por *Kraitchik* (1926); véase también *Pomerance* (1982). La historia del algoritmo de factorización de criba cuadrática se da en *Pomerance* (1984) y la variante doblemente prima utilizada para afrontar el desafío RSA procede de *Lenstra* y *Manasse* (1991). El algoritmo de factorización basado en curvas elípticas se discute en *Lenstra* (1987). La criba de cuerpos numéricos se describe en *Lenstra, Lenstra; Manasse* y *Pollard* (1993). Véase tam-

bién *Koblitz* (1987) y *Bressoud* (1989). La técnica para buscar en una lista ordenada proviene de *Janko* (1986); véase el problema 10.36. Se da un análisis detallado de esta técnica en *Bentley, Stanat* y *Steele* (1981), donde se demuestra además que se necesita un tiempo esperado en $\Omega(\sqrt{n})$ en el caso peor para resolver este problema por cualquier algoritmo probabilista.

Hay varios algoritmos probabilistas interesantes que no hemos tratado en este libro. Concluiremos mencionando algunos de ellos. Dadas las coordenadas cartesianas de puntos del plano, *Rabin* (1976) proporciona un algoritmo capaz de hallar el par más próximo en un tiempo esperado lineal; compárese esto con el problema 7.39. En *Schwartz* (1978) se da un algoritmo de Monte Carlo para decidir si un polinomio en varias variables sobre un dominio infinito es idénticamente cero, y para comprobar si dos de estos polinomios son idénticos. Consulte *Zippel* (1979) para ver algoritmos de probabilistas de interpolación en polinomios dispersos. *Rabin* (1980a) da un algoritmo probabilista eficiente para calcular raíces de polinomios arbitrarios en cualquier cuerpo finito, así como un algoritmo probabilista eficiente para factorizar polinomios en cuerpos finitos arbitrarios, y para hallar polinomios irreducibles. Hay un algoritmo de Las Vegas muy elegante para buscar raíces cuadradas módulo en un número primo; se debe a *Peralta* (1986); véase también *Brassard* y *Bratley* (1988). Un ejemplo raro de algoritmo de Monte Carlo no sesgado para un problema de decisión, que puede decidir eficientemente si un entero dado es un número perfecto y si una pareja de enteros es amigable, se describe en *Bach, Miller* y *Shallit* (1986).

Capítulo 11

Algoritmos paralelos

En otras partes de este libro suponemos implícitamente que nuestros algoritmos se ejecutarán en una máquina que sólo puede ejecutar una instrucción a la vez. Por supuesto, cualquier máquina moderna simultanea el cálculo con las operaciones de entrada/salida tales como la espera a que se pulse una tecla, o a que se imprima un archivo. Muchas de ellas simultanean también distintas operaciones aritméticas cuando se calcula una expresión, de tal manera que las sumas, por ejemplo, se pueden efectuar en paralelo con las multiplicaciones. Sin embargo, hasta el momento no hemos considerado la posibilidad de que la máquina pudiera calcular varias docenas, o incluso varios centenares de expresiones diferentes a la vez. Si consideramos esta posibilidad, entonces podemos tener la esperanza, si somos a la vez inteligentes y afortunados, de acelerar algunos de nuestros algoritmos en la misma medida.

Las computadoras capaces de efectuar estos cálculos en paralelo no se encuentran todavía en la mesa de cualquier despacho. Sin embargo, su número crece, y el interés por los *algoritmos paralelos*, que aprovechan esta capacidad, está muy extendido. La investigación en este sentido es tan activa que no sería realista intentar mencionar todos los campos en los que se están estudiando técnicas paralelas. Consiguientemente, en este capítulo presentaremos solamente una selección introductoria de algoritmos paralelos que ilustran algunas técnicas fundamentales.

En primer lugar, describiremos con más precisión la máquina en que estamos pensando cuando se diseñan estos algoritmos. A continuación, ilustraremos una o dos técnicas básicas, y discutiremos lo que queremos decir al hablar de un algoritmo paralelo eficiente. Por último, daremos un pequeño número de ejemplos pertenecientes a los campos de teoría de grafos, de evaluación de expresiones y de ordenación.

11.1 UN MODELO PARA LA COMPUTACIÓN PARALELA

El modelo básico para la computación ordinaria, secuencial, en lo que a veces se llama una *máquina de von Neumann*, está aceptado tan ampliamente que no nos ha parecido necesario definirlo con más precisión en este libro. Todo el

mundo admite que estas máquinas ejecutan una instrucción cada vez, aplicándola a un dato cada vez, y siguiendo un programa que está almacenado en la memoria de la máquina. El hecho de que, en realidad, todas las máquinas modernas poseen un cierto grado de paralelismo, que las capacita por ejemplo para efectuar un número limitado de operaciones aritméticas en paralelo, tomar la instrucción siguiente mientras se está ejecutando la última, o simultanear la entrada y la salida con el cómputo, es esencialmente irrelevante para el modelo conceptual. De manera similar, la mayoría de los lenguajes de programación, como el lenguaje informal que empleamos en este libro, suponen que la computadora ejecuta una instrucción cada vez, y que se la aplica a una sola variable.

En el campo de la computación en paralelo, las máquinas operativas son relativamente pocas, y sus arquitecturas variadas, y no existe un consenso en lo tocante a cuál de los modelos teóricos de computación es el mejor de todos. Por ejemplo, no es evidente lo que podría implicar en la práctica seleccionar un modelo que permita a varios procesadores asignar nuevos valores a una misma variable en paralelo. Las restricciones prácticas evitarán, probablemente, que estas asignaciones en paralelo se efectúen realmente al mismo tiempo. ¿Debemos entonces suponer que ahora la variable tiene el último valor que se le haya asignado? En tal caso ¿sabemos cuál ha sido esa última asignación? O si son *realmente* posibles las asignaciones simultáneas, ¿cuál es el resultado de asignar simultáneamente varios valores a la misma variable? Estas consideraciones llevan a algunas personas a preferir un modelo que prohíba las asignaciones simultáneas múltiples a la misma variable.

Una vez más, supongamos que tenemos un vector que va a ser modificada por un cierto número de procesadores paralelos, con un procesador para cada elemento del vector. Si deseamos modelar una instrucción tal como «detenerse cuando sean nulos todos los elementos del vector», ¿cuánto tiempo debemos suponer que es necesario para implementar esta instrucción? Ciertamente, cada procesador puede ver inmediatamente que su propio elemento es cero, pero ¿cuál debemos de suponer que es el mecanismo necesario para comprobar *todos* los elementos? ¿Tienen los procesadores que intercambiar mensajes entre sí, hay una capa adicional de procesadores para el control global, o cómo se llega exactamente a una decisión?

Por estas razones, y por otras similares, no hay una respuesta clara para la pregunta acerca de qué modelo de computación en paralelo es el mejor en general. En este libro utilizamos un modelo popular llamado *máquina paralela de acceso aleatorio*, o *p-ram*. Este modelo es, ciertamente, el de uso más natural, y resulta fácil de entender, pero no es excesivamente similar a las máquinas existentes. Quede advertido el lector, por tanto, de que puede no resultar sencillo adaptar los algoritmos de este capítulo para su utilización en una máquina real. La Sección 11.11 contiene indicaciones de computadoras paralelas más realistas.

En el modelo *p-ram*, se supone que hay procesadores secuenciales que comparten una memoria global. Cada procesador posee el conjunto habitual de

instrucciones aritméticas y lógicas, instrucciones que puede ejecutar en paralelo con lo que esté sucediendo en los demás procesadores. Sin embargo, suponemos que los procesadores no se ponen a hacer cosas distintas, sino que todos ellos ejecutan un mismo programa que se les suministra desde algún punto central de control, aunque posiblemente con distintos datos. Por razones evidentes, este modelo se llama también de *instrucción única y múltiples flujos de datos*. Por el momento, no definiremos con más precisión lo que queremos decir con esto, y nos basaremos en ejemplos para clarificar una idea que en lo básico es sencilla.

Todo procesador tiene acceso a toda la memoria global. En cada paso, puede o bien leer o bien escribir en una sola posición de memoria. A nuestros efectos, supondremos además que aun cuando de una posición de memoria puedan leer cuantos procesadores lo deseen en un mismo instante, en esa posición no pueden escribir dos procesadores simultáneamente, ni tampoco es posible que un procesador lea una posición de memoria en la que se está escribiendo en el mismo instante. Así evitamos tener que decidir lo que sucede si dos o más procesadores intentan escribir valores diferentes en la misma posición, o lo que pasa si cambia un valor mientras un procesador lo está leyendo. Los modelos definidos de esta manera se llaman modelos de *lectura concurrente y escritura exclusiva*, o CREW[1]. Otros posibles modelos, que no consideraremos por el momento, son el EREW (lectura exclusiva, escritura exclusiva) y el CRCW (lectura concurrente, escritura concurrente), que se definen de la forma natural. Nadie parece haber encontrado aplicaciones para un modelo de lectura exclusiva y escritura concurrente.

Cuando se analicen algoritmos paralelos en las secciones siguientes, haremos una suposición crucial: que el acceso a memoria en nuestro hipotético *p-ram* CREW, sea para leer o para escribir, se puede hacer en un tiempo constante, independientemente del número de procesadores utilizados. Esta suposición no es cierta en la práctica. Dado que no es factible proporcionar accesos directos en *hardware* desde todos los procesadores hasta todas las posiciones de memoria, el tiempo medio necesario para efectuar un acceso a memoria en un sistema real crece a medida que asciende el número de procesadores; además, algunos esquemas de acceso a memoria son más rápidos que otros. De hecho, ni siquiera es cierto que un único procesador pueda acceder a todas las direcciones de un espacio arbitrariamente grande en un tiempo constante. Por sencillez, sin embargo, en este libro ignoraremos esta complicación.

Por sencillez, también, ignoraremos la mayoría de los problemas planteados por el control global de la máquina paralela. Para describir nuestros algoritmos paralelos, utilizamos instrucciones de la forma general

para $x \in S$ **hacer en paralelo** *instrucción*(x)

Esto quiere decir que asignamos un procesador a cada elemento x del conjunto S, y que entonces se ejecutan las instrucciones de *instrucción*(x) para cada uno de

estos elementos en paralelo, utilizando x como datos, y calculando en el procesador asignado. Suponemos que los procesadores se numeran en secuencia, y que los elementos del conjunto S también se pueden numerar de forma sencilla, para que la asignación de un procesador concreto a x se pueda hacer en un tiempo constante. Se utilizarán variaciones de esta forma de instrucción sin más explicaciones si está claro su significado. Por ejemplo:

para $1 \leq i \leq 10$ **hacer en paralelo** *instrucción*(i)

significa que hay que ejecutar *instrucción*(i) en paralelo para $i = 1, 2, ..., 10$.

Hay que tener cuidado para que la *instrucción* que es preciso ejecutar respete nuestro requisito (bastante impreciso por el momento) de que todos los procesadores «ejecutan la misma instrucción al mismo tiempo». Por ejemplo, no permitimos que *instrucción* sea una llamada a una función, porque esto podría dar lugar a acciones bastante diferentes por parte de cada procesador. Sin embargo, las instrucciones de asignación, los accesos a matrices y demás son aceptables en general. También admitiremos instrucciones condicionales, pero no bucles cuya longitud dependa de los datos. De esta manera, nuestro requisito va estando más claro: al decir «la misma sentencia» queremos decir más o menos «la misma instrucción de máquina», y no «la misma sentencia en algún lenguaje de alto nivel». Aceptaremos, sin embargo, que en el caso de una instrucción condicional, por ejemplo, algunos de los procesadores puedan ejecutar la instrucción aunque otros la pasen por alto.

Si un cálculo necesita utilizar p procesadores, podemos suponer además que se necesita un tiempo que está en $\Theta(\log p)$ previo al comienzo de la computación, para enviarles las instrucciones necesarias y hacer que empiecen a trabajar. Esto se consigue fácilmente si inicialmente está activo un procesador, y después en cada paso temporal cada procesador activo activa a otro más, de tal modo que el número de procesadores activos se doble a cada paso. Sin embargo, en los párrafos siguientes no tomamos en cuenta este tiempo de inicialización en forma explícita.

11.2 TÉCNICAS BÁSICAS

11.2.1 Cómputo con un árbol binario completo

La mejor manera de explicar esta sencilla técnica es a través de un ejemplo. Supongamos que deseamos calcular la suma de n enteros. Para evitar complicaciones, supongamos también que n es una potencia de 2; de no ser así, basta con añadir los elementos nulos, mudos, que sean necesarios. Estos n elementos se sitúan en las hojas de un árbol binario completo, según se ilustra en la figura 11.1. Ahora, en el primer paso, se calculan en paralelo las sumas de los elementos de nivel 1; en el segundo paso, se calculan en paralelo las sumas de los elementos del nivel 2; y así sucesivamente hasta que en el paso (lg n)-ésimo el valor que se obtiene en la raíz es la solución de nuestro problema.

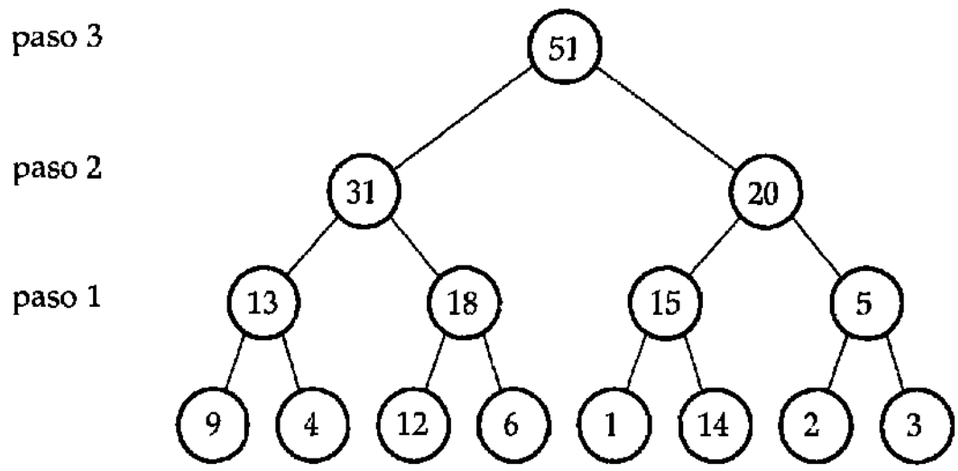


Figura 11.1. Cálculo con un árbol binario completo

Más formalmente, sea $n = 2^k$, y sea T un vector con índices en el rango 1 a $2n-1$. Este vector se puede emplear para almacenar un árbol binario completo con $2n-1$ nodos, con la raíz en $T[1]$ y con los hijos del nodo $T[i]$ en los nodos $T[2i]$ y $T[2i+1]$, exactamente igual que un montículo. Supongamos que los n elementos que hay que sumar están situados inicialmente en las hojas del árbol, esto es, en los nodos que van desde $T[n]$ hasta $T[2n-1]$. Ahora, el algoritmo para calcular la suma de estos n elementos es como sigue:

```
función sumapar(T,n)
    {Calcula la suma T[n]+...+T[2n−1]}
    para i ← lg n −1 bajando −1 hasta 0 hacer
        para 2^i ≤ j ≤ 2^(i+1) − 1 hacer en paralelo
            T[j] ← T[2j] + T[2j+1]
    devolver T[1]
```

En este caso la única sincronización que se requiere es que todos los cálculos paralelos para un valor particular de i deben haber finalizado antes de que comiencen los cálculos correspondientes al próximo valor de i. Durante una pasada por el bucle i, cada procesador implicado efectúa dos accesos a memoria para leer sus operandos, una adición y un acceso final a memoria para almacenar el resultado. Dado que suponemos que un acceso a memoria requiere un tiempo constante, el trabajo total necesario por parte de cada procesador también requiere un tiempo constante. Por último, dado que los procesadores funcionan en paralelo, todo el trabajo necesario para una pasada por el bucle se puede ejecutar en un tiempo constante. El algoritmo efectúa lg n pasadas por el bucle, así que el tiempo total requerido está en $\Theta(\log n)$. Es evidente que el máximo número de procesadores que tienen que funcionar simultáneamente es $n/2$.

Está claro que se puede aplicar la misma técnica a problemas tales como buscar el producto, el máximo o el mínimo de n elementos, o para decidir si son todos nulos.

11.2.2 Duplicación de punteros

El algoritmo de duplicación de punteros, en su forma más sencilla, puede aplicarse a listas de elementos. Supongamos que se dispone de una lista L de n elementos, cada uno de los cuales contiene un puntero de su sucesor. Supongamos que el sucesor del elemento i es $s[i]$. Si el elemento k es el último elemento de la lista, entonces $s[k]$ es el puntero especial **nil**. Como primer ejemplo, queremos calcular para cada elemento i la distancia $d[i]$ desde ese elemento hasta el final de la lista. Suponemos que están disponibles tantos procesadores como elementos haya en L, así que podemos asociar un procesador distinto a cada elemento de la lista. Ahora el algoritmo es como sigue:

```
procedimiento distpar(L)
    { Iniciación }
    para cada elemento i ∈ L hacer en paralelo
        si s[i] = nil entonces d[i] ← 0
                          sino d[i] ← 1
    { Bucle principal }
    repetir ⌈lg n⌉ veces
        para cada elemento i ∈ L hacer en paralelo
            si s[i] ≠ nil entonces
              d[i] ← d[i] + d[s[i]]
              s[i] ← s[s[i]]
```

La figura 11.2 ilustra el progreso de este algoritmo para una lista de 7 elementos. Tal como está escrito, los campos de punteros de la lista original se modifican, y se destruye la estructura de la lista. Si no se desea esto, se copian los punteros en la fase de iniciación del algoritmo, y después se trabaja con las copias.

Aquí la sincronización es más sutil que en el ejemplo anterior. No hay problema con los intentos simultáneos de escribir en una misma posición, puesto que cada procesador asigna solamente valores a d y a s en su propio elemento. En el modelo que hemos adoptado, no es preciso preocuparse por los intentos simultáneos de leer en la misma posición. Sin embargo, *sií* hay que preocuparse por los valores que cambian antes de que hayamos tenido tiempo de leerlos. Cuando varios procesadores están ejecutando la instrucción $d[i] \leftarrow d[i] + d[s[i]]$ en paralelo, por ejemplo, la sincronización debe ser tan fuerte que nos permita asegurar que el procesador asignado al elemento i va a leer el valor necesario de $d[s[i]]$ antes de que el procesador asignado al elemento $s[i]$ cambie su valor de d. La forma más segura de asegurar esto para todo i es insistir en que *todas* las lecturas necesarias para evaluar las expresiones del lado derecho se ejecuten antes de haber escrito ningún valor nuevo en las variables del lado izquierdo. De manera similar, en la instrucción $s[i] \leftarrow s[s[i]]$, las lecturas necesarias para evaluar las expresiones del lado derecho deben de producirse todas ellas antes de que se haya escrito ningún valor

nuevo en las variables del lado izquierdo. El requisito enunciado anteriormente, de que todos los procesadores estén trabajando en la misma instrucción al mismo tiempo, tiene que interpretarseahora de manera más estricta, con sincronización a nivel de las instrucciones de máquina.

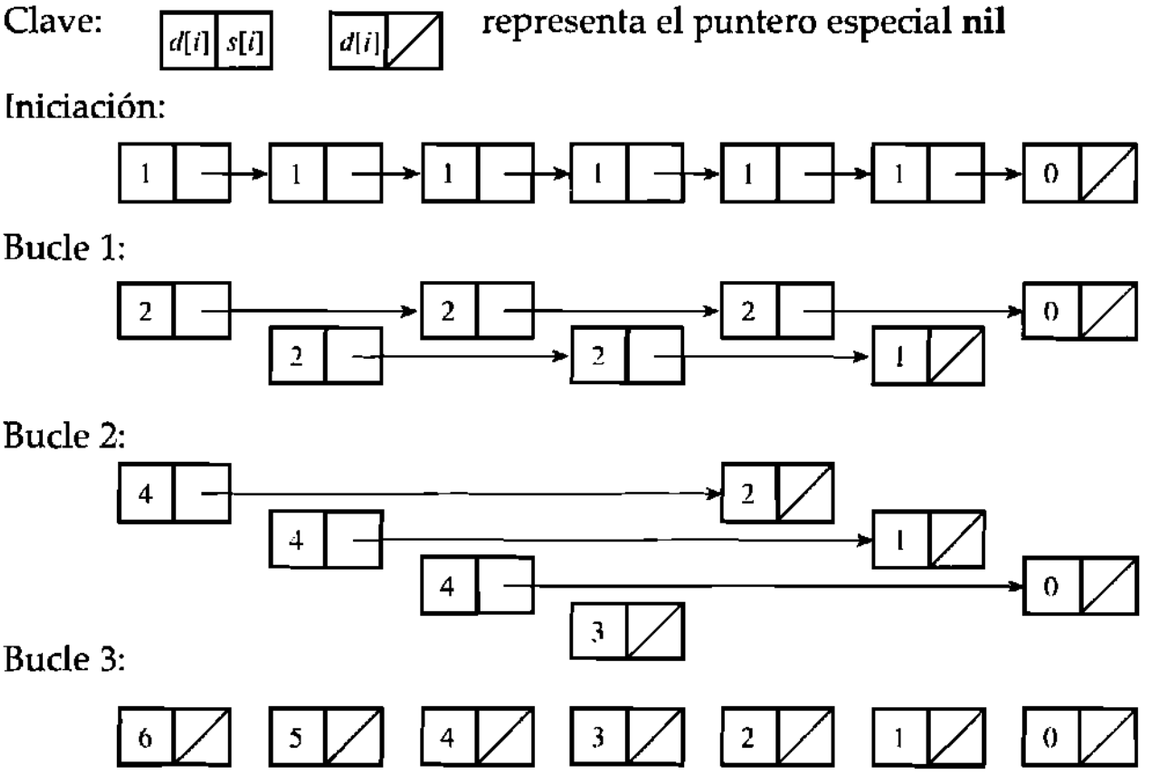


Figura 11.2. Duplicación de punteros

Cuando se detiene el algoritmo, $s[i]$ = **nil** para todo elemento i de L. Para ver esto, observe que los punteros se "duplican" en cada ejecución de la instrucción $s[i] \leftarrow s[s[i]]$. Más exactamente, $s[i]$ apunta originalmente al elemento que sigue a i dentro de L; después de una ejecución de esta instrucción, $s[i]$ apunta al elemento que originalmente estaba a dos lugares de distancia de i; después de dos ejecuciones de la instrucción, apunta al elemento que originalmente estuviera a cuatro lugares de i, y así sucesivamente. Cuando un puntero se sale de la lista le damos el valor especial **nil**. Dado que en L hay n elementos, basta duplicar la longitud de los punteros $\lceil \lg n \rceil$ veces para asegurarnos de que todos se «salen» por un extremo (para el caso en que n sea desconocido, véase el problema 11.21).

Para ver que los valores calculados de $d[i]$ son correctos, observe que al principio de cada iteración, si sumamos los valores de d para todos los elementos de la sub-lista encabezada por el elemento i (empleando, por supuesto, los valores actuales de s), obtenemos la distancia desde i hasta el final de la lista original L. Ahora, en cada iteración el puntero $s[i]$ se modifica de tal manera que se omita el sucesor inmediato de i en esta sub-lista. Sin embargo, el valor de d para este sucesor inmediato se suma a $d[i]$, así que sigue siendo cierta la misma condición al principio de la iteración siguiente.

Manteniendo nuestra suposición de que todos los accesos a memoria requieren un tiempo constante, vemos inmediatamente que el trabajo requerido por parte de cada procesador en una iteración del bucle **repetir** es constante, así que el tiempo de ejecución para el algoritmo completo está en $\Theta(\log n)$. Hay un procesador por cada elemento de la lista, así que el número total de procesadores requeridos es n.

Si la estructura de datos original no es una lista, sino una estructura de partición (véase la Sección 5.9), se puede aplicar un algoritmo similar. Supongamos que se representa cada conjunto mediante un árbol, tal como en la figura 5.21, con punteros que van desde cada elemento hasta su predecesor, salvo que las raíces se apuntan a sí mismas. Ahora se aplica un algoritmo similar a *distpar* salvo que omite toda mención a las distancias d; este algoritmo «aplana» los árboles de tal manera que todos los elementos señalen directamente a la raíz adecuada. Se puede ver esto como un ejemplo extremo de compresión de rutas. El algoritmo es el siguiente.

```
procedimiento aplanar(d)
    {D es una estructura de subconjunto disjunto con n elementos}
    repetir ⌈lg n⌉ veces
        para todo elemento i ∈ D hacer en paralelo
            s[i] ← s[s[i]]
```

Se utilizan variantes del método de duplicación de punteros para muchos otros cálculos sencillos. Supongamos por ejemplo que todo elemento i de la lista L tiene un valor $v[i]$, y sea $\circ$ un operador binario asociativo que toma como operandos estos valores. Considere la variante siguiente del procedimiento *distpar*.

```
procedimiento operpar(L)
    {Inicialización}
    para todo elemento i ∈ L hacer en paralelo
        d[i] ← v[i]
    {Bucle principal }
    repetir ⌈lg n⌉ veces
        para todo elemento i ∈ L hacer en paralelo
            si s[i] ≠ nil entonces
                d[i] ← d[i]∘d[s[i]]
                s[i] ← s[s[i]]
```

Sea x_i el valor de v para el i-ésimo elemento de la lista, con $1 \le i \le n$. (Esto *no* es lo mismo que $v[i]$). En el caso de x_i, el subíndice indica que deseamos el valor del i-ésimo elemento de la lista original; en el caso de $v[i]$, el índice es un puntero a un elemento, y no su posición dentro de la lista.) Se define $X_{i,j}$ mediante

$$X_{ij} = x_i \circ x_{i+1} \circ \cdots \circ x_j ,$$

esto es, como el producto generalizado de los elementos de L desde el i-ésimo hasta el j-ésimo. Entonces no resulta difícil probar (véase el Problema 11.3) que cuando termina el algoritmo, el valor de d para el primer elemento de L es $X_{1,n}$, el valor de d para el segundo elemento de L es $X_{2,n}$, y así sucesivamente, hasta el último elemento de L cuyo valor de d es $X_{n,n} = x_n$. La figura 11.3 ilustra el funcionamiento del algoritmo para una lista de siete elementos en la cual el operador ∘ es la suma habitual.

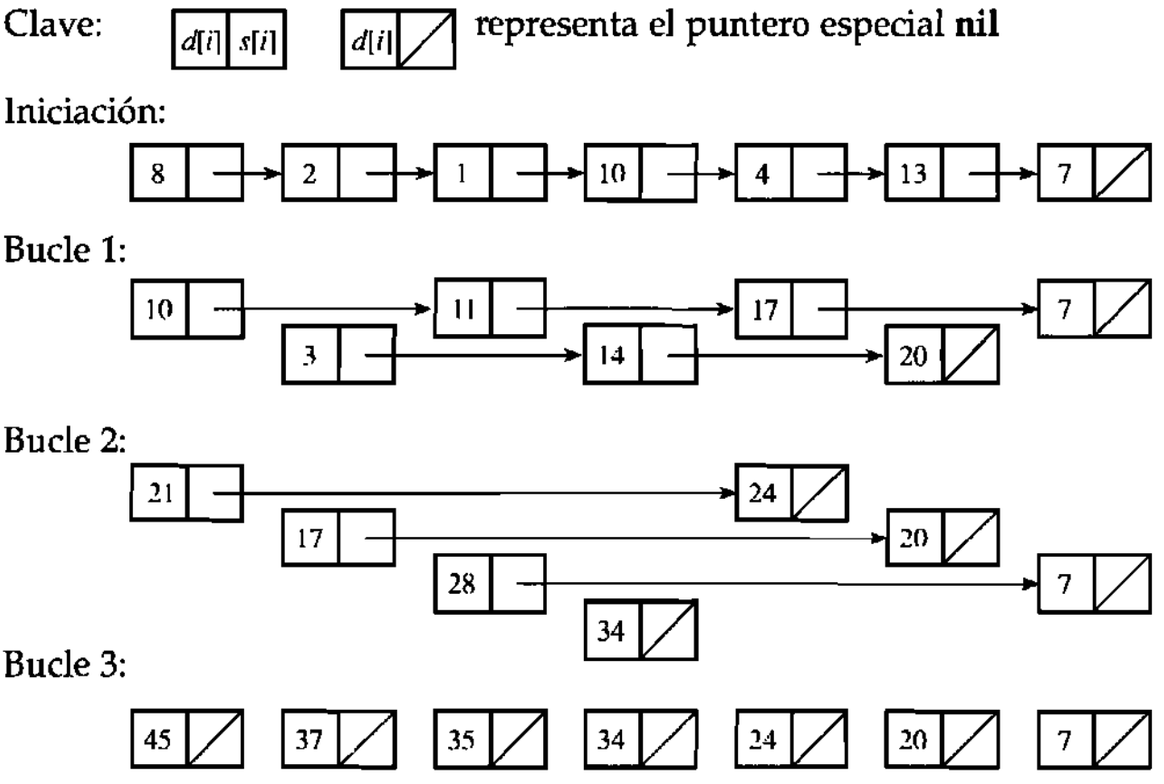


Figura 11.3. El algoritmo operpar

Seleccionando un operador adecuado, se pueden efectuar varios cálculos de utilidad. Tomando el operador ∘ como operador suma, tal como en el ejemplo de la figura 11.3, se nos da la suma de los elementos de L. Cuando se detiene el algoritmo, se proporciona la suma como valor de d para el primer elemento de L. El producto de los elementos de L, su máximo o su mínimo se pueden calcular de manera semejante. También son posibles unos esquemas más imaginativos. Por ejemplo, si cada $d[i]$ es inicializado con un puntero al propio elemento i en sí, y ∘ es un operador tal que al recibir dos punteros proporciona un puntero del elemento que tenga el mayor valor de v, entonces *operpar* se puede utilizar para obtener un puntero al elemento de L que tenga el mayor valor, y no solamente el valor en sí. Para un mayor desarrollo de este tema, véase el problema 11.5.

El análisis de *operpar* es exactamente igual que el de *distpar*. El tiempo de ejecución del algoritmo, siempre y cuando el operador ∘ se pueda tratar como elemental, está en $\Theta(\log n)$, y el número de procesadores necesarios es n.

11.3 TRABAJO Y EFICIENCIA

Parece probable que un algoritmo paralelo que tarde t segundos en ejecutarse con digamos 20 procesadores efectue más cálculos que uno que tarde el mismo tiempo empleando sólo cinco procesadores, que a su vez trabaja más que un algoritmo secuencial ordinario que también requiera t segundos. Por esta razón, definimos el *trabajo* efectuado por un algoritmo, tanto si es secuencial como si es paralelo, como el producto pt del número p de procesadores que utiliza por el tiempo de ejecución t. (Dado que para muchos algoritmos el número de procesadores necesarios varía durante el cómputo, se puede argumentar que el trabajo efectuado se mediría mejor mediante $\sum it_i$, en donde t_i es el tiempo durante el cual están activos exactamente i procesadores). Sin embargo, esto supone implícitamente que los procesadores no utilizados quedan libres para hacer alguna otra cosa. En todo caso, es demasiado complicado para lo que necesitamos en este momento). Se ve fácilmente que para un algoritmo paralelo este trabajo es el tiempo necesario para simular el algoritmo paralelo empleando un único procesador, que en cada paso del cómputo imita por turno a cada uno de los procesadores paralelos. Si tenemos dos algoritmos A y B para un mismo problema, que requieren trabajos w_a y w_b respectivamente para obtener una solución, diremos que A es *eficiente en trabajo* con respecto a B si $w_a \in O(w_b)$.

En la Sección 12.5.1 veremos que un algoritmo ordinario, secuencial, se suele considerar eficiente si su tiempo de ejecución para un problema de tamaño n está en $O(n^k)$ para alguna constante k. Por otra parte, para que se considere eficiente un algoritmo paralelo, esperamos normalmente que satisfaga dos restricciones, una tocante al número de procesadores y otra que concierne al tiempo de ejecución. Ha de cumplir:

- el número de procesadores necesarios para resolver un caso de tamaño n debe estar en $O(n^a)$ para alguna constante a, y
- el tiempo requerido para resolver un caso de tamaño n debe estar en $O(\log^b n)$ para alguna constante b.

Diremos que un algoritmo paralelo eficiente requiere un número polinómico de procesadores, y un tiempo *polilogarítmico.*

Un algoritmo paralelo se dice *óptimo* si es eficiente en trabajo respecto el mejor algoritmo secuencial posible. A veces se puede denominar óptimo si es tarea eficiente con respecto al mejor algoritmo secuencial *conocido.* En este caso, sin embargo, es preferible decir que el problema correspondiente tiene *aceleración óptima.* Veremos en el Capítulo 12 que hay muchos problemas para los cuales no se conoce ningún algoritmo secuencial eficiente (esto es, de tiempo polinómico). Para tales problemas, no podemos esperar hallar una solución paralela eficiente (esto es, una que utilice un número polinómico de procesadores y un tiempo polilogarítmico): véase el problema 11.6. Por otra parte, hay muchos problemas para los cua-

les se conoce un algoritmo secuencial eficiente, pero para los cuales todavía no se ha descubierto un algoritmo paralelo eficiente. Se cree, aunque no se ha demostrado, que algunos problemas que se pueden resolver mediante un algoritmo secuencial eficiente no poseen una solución paralela eficiente.

Considérese la técnica descrita en la Sección 11.2.1 como ejemplo. Vimos allí que se puede calcular la suma de n elementos almacenados en un vector empleando $n/2$ procesadores y utilizando un tiempo que está en $\Theta(\lg n)$. El trabajo que requiere el algoritmo está por tanto en $(n/2)\times\Theta(\log n)=\Theta(n\log n)$. Puesto que la suma de n elementos se puede obtener claramente en $\Theta(n)$ operaciones por simple adición en un procesador secuencial, el algoritmo paralelo, aunque es eficiente, no es óptimo. De manera similar, las técnicas descritas en la Sección 11.2.2 nos permiten efectuar toda una variedad de operaciones (calcular la distancia hasta el extremo de una lista de n elementos, hallar la suma o el máximo de la lista de elementos, etc.) en un tiempo que está en $\Theta(\log n)$ empleando n procesadores. Una vez más, el trabajo necesario está en $\Theta(n\log n)$. Puesto que estas operaciones se pueden efectuar mediante un solo procesador secuencial en un tiempo que está en $\Theta(n)$, estos algoritmos tampoco son óptimos.

Si examinamos con más detalle el algoritmo necesario para calcular la suma de n elementos empleando un árbol binario, una posible razón para que no llegue a ser óptimo es patente a primera vista. En la primera pasada por el bucle principal, se necesitan $n/2$ procesadores, y esto determina los recursos necesarios para el algoritmo, porque en la segunda pasada por el bucle sólo hay $n/4$ procesadores que realicen un trabajo útil; en la tercera sólo se necesitan $n/8$ y así sucesivamente: durante la mayor parte del tiempo, casi todos los procesadores están ociosos. Esto sugiere que quizá seamos capaces de utilizar menos procesadores sin que esto tenga un efecto catastrófico sobre el tiempo de cómputo.

Supongamos que sólo están disponibles $p < n/2$ procesadores. Una forma de proceder consiste en dividir en p grupos los números cuya suma haya que calcular; $p-1$ grupos contienen n/p números, mientras que el último contiene los $n-(p-1)\lceil n/p\rceil$ números. El último grupo, por tanto, puede contener menos de $\lceil n/p\rceil$ miembros, pero nunca puede contener más. Ahora asignamos a cada grupo uno de los procesadores disponibles, y ponemos a cada procesador a calcular la suma de su grupo. Aun cuando los procesadores trabajan en paralelo, los cálculos individuales pueden ser simples cómputos secuenciales que requerirán $\Theta(\lceil n/p\rceil)$ operaciones. Dado que los procesadores trabajan en paralelo, el tiempo total requerido para esta fase también está en $\Theta(\lceil n/p\rceil)$. El problema se reduce ahora a hallar la suma de las p sumas de grupo, y esto se puede resolver mediante la técnica del árbol equilibrado sin modificación empleando $p/2$ procesadores en un tiempo que está en $\Theta(\log p)$. En total, el algoritmo modificado que emplea $p < n/2$ procesadores requiere un tiempo que está en $\Theta(\lceil n/p\rceil+\log p)$.

En particular, si hacemos $p = n/\log n$ obtenemos un algoritmo que puede calcular la suma de n números en un tiempo que está en $\Theta(\log n)$. Por tanto hemos reducido el número de procesadores necesarios por un factor $(\log n)/2$, sin modi-

ficar el orden del tiempo de ejecución. El trabajo efectuado por el algoritmo modificado está en $\Theta(p \times \log n) = \Theta(n)$. Es claro que ningún algoritmo secuencial puede hacerlo mejor que esto, así que el algoritmo modificado es un algoritmo paralelo óptimo.

En general, podemos no ser tan afortunados. Por ejemplo, dividir los elementos de una lista en grupos es más difícil que dividir los elementos de un vector. Para la primera, podemos tener que empezar por explorar toda la lista; para el segundo, suele ser suficiente un cálculo sencillo haciendo uso de los índices del vector. Sin embargo, empleando una técnica similar siempre podemos reducir el número de procesadores requeridos por un algoritmo paralelo. Supongamos que tenemos un algoritmo que funciona en un tiempo t empleando p procesadores para un problema de tamaño n, pero que sólo disponemos de $q < p$ procesadores (aquí, t, p y q son funciones de n). ¿Qué deberíamos hacer?

Tal como antes, dividimos los p procesadores en q grupos, y utilizamos uno de los q procesadores disponibles para simular cada grupo. Habrá $q-1$ grupos que contengan p/q procesadores, y al menos un grupo que no contendrá más procesadores que los otros, y quizá menos. A continuación, supongamos que el algoritmo original efectúa los pasos 1,2,..., en donde los p procesadores ejecutan independientemente cada paso, pero es preciso sincronizarlos entre pasos. En el algoritmo modificado que emplea solamente q procesadores, en el paso primero uno de ellos simula por turnos a cada procesador del primer grupo; un segundo procesador simula por turnos a cada procesador del segundo grupo, y así sucesivamente. Dado que hay p/q procesadores, o menos, que simular en cada grupo, la simulación del paso 1 empleando q procesadores requiere p/q veces más que en el paso 1 original, y lo mismo sucede para los demás pasos. De esta manera, el cálculo completo empleando q procesadores requiere p/q veces más tiempo que el cálculo completo empleando p procesadores. En símbolos, el algoritmo modificado requiere un tiempo que está en $\Theta(\lceil p/q \rceil t)$. (Recuerde que p, q y t pueden ser funciones del tamaño n del caso.) Dado que $p/q \leq \lceil p/q \rceil < 2p/q$ cuando $p > q$, hemos demostrado el teorema siguiente.

Teorema 11.3.1 (Brent) *Si existe un algoritmo paralelo que requiere un tiempo $t(n)$ para resolver un problema de tamaño n empleando $p(n)$ procesadores, entonces para cualquier $q(n) < p(n)$ existe un algoritmo modificado que puede resolver el mismo problema empleando solamente $q(n)$ procesadores en un tiempo $\Theta(p(n)t(n)/q(n))$.*

Aquí, el algoritmo original funciona ciertamente en $\Theta(p(n)t(n)/q(n) \times q(n)) = \Theta(p(n)t(n))$, así que en términos de trabajo no hemos ganado ni perdido con la modificación: el algoritmo modificado es eficiente en trabajo con respecto al algoritmo original. Por tanto, podemos reducir el número de procesadores empleados por un algoritmo sin reducir su eficiencia. En particular, si el algoritmo original es óptimo, entonces también lo es el algoritmo modificado. Por supuesto, el algorit-

mo que utilice menos procesadores tardará normalmente más tiempo en terminar, aun cuando el trabajo efectuado sea el mismo.

11.4 DOS EJEMPLOS DE TEORÍA DE GRAFOS

11.4.1 Caminos mínimos

Ya hemos encontrado variantes de este problema en las Secciones 6.4 y 8.5. Repetiremos brevemente los detalles. Sea $G = <N, A>$ un grafo dirigido. Los nodos de G están numerados de 1 a n, y hay una matriz L que da las longitudes de las aristas, con $L[i, i] = 0$; $L[i, j] \geq 0$ si $i \neq j$, y $L[i, j] = \infty$ si no existe la arista (i, j). Deseamos calcular la longitud de el camino más corto entre todos los pares de nodos. Ahora daremos un algoritmo paralelo para este problema. Es interesante comparar esto con el algoritmo *Floyd* de la Sección 8.5.

El algoritmo paralelo construye una matriz D que da la longitud del camino más corto entre todos los pares de nodos. Da a D el valor inicial L, esto es, las distancias directas entre nodos, y después efectúa $\lceil \lg n \rceil$ iteraciones. Después de la iteración k, D da la longitud de los caminos mínimos que no utilizan más de 2^k aristas, o equivalentemente, que no utilizan más de $2^k - 1$ nodos intermedios.

Como las longitudes de las aristas son no negativas, los caminos mínimos que estamos buscando deben de ser simples: esto es, no pueden visitar dos veces el mismo nodo. Por tanto, utilizan como máximo $n - 1$ aristas. Consiguientemente, al cabo de $\lceil \lg n \rceil$ iteraciones, D da el resultado deseado.

En la iteración k, el algoritmo debe de comprobar para cada pareja de nodos (i,j) si existe o no una nuevo camino desde i hasta j que utilice más de 2^{k-1} aristas pero menos de 2^k, y que sea mejor que el presente camino óptimo, que no utiliza más de 2^{k-1} aristas. Cualquiera de estas caminos nuevos tiene un nodo «medio»llamado m definido de tal manera que ni la parte de el camino que va de i a m ni la que va de m a j utilicen más de 2^{k-1} aristas. Las longitudes óptimas de estas partes son por tanto los valores actuales de $D[i, m]$ y de $D[m, j]$, respectivamente. Dado que es aplicable el principio de optimalidad, para comprobar si existe un nuevo camino mejorado desde i hasta j, basta comparar la longitud del mejor camino existente con $D[i, m] + D[m, j]$ para todos los posibles valores de m. Entonces nos limitamos a seleccionar el mínimo. Véase el algoritmo.

```
procedimiento caminospar(L[1..n, 1..n]): matriz[1..n, 1..n]
    matriz D[1..n, 1..n], T[1..n, 1..n, 1..n]
    para todo i,j hacer en paralelo D[i, j] ← L[i, j]
    repetir ⌈lg n⌉ veces
        para todo i, j, m hacer en paralelo
            T[i, m, j] ← D[i, m] + D[m, j]
        para todo i,j hacer en paralelo
            D[i, j] ← min(D[i, j], T[i,1, j], T[i, 2,j],...,T[i, n, n])
    devolver D
```

Aquí la matriz T contiene las longitudes de los caminos para evitar conflictos entre lecturas y escrituras en la última instrucción **para**. No hay conflicto entre la lectura del viejo valor de $D[i, j]$ y la escritura de su nuevo valor en esta instrucción, puesto que esto es efectuado por el mismo procesador. Las variables i, j y m varían entre 1 y n.

El análisis de este algoritmo es sencillo. La primera instrucción **para** se puede ejecutar en un tiempo constante empleando n^2 procesadores. Dentro de la instrucción **repetir**, la primera instrucción **para** se puede ejecutar en tiempo constante empleando n^3 procesadores. Se puede calcular el mínimo de entre $n+1$ elementos en un tiempo que está en $\Theta(\log n)$ empleando $\Theta(n / \log n)$ procesadores, según se describe en la Sección 11.2.1. Hay que calcular n^2 de estos mínimos en paralelo, así que una iteración de la segunda instrucción **para** se puede ejecutar en un tiempo que está en $\Theta(\log n)$ empleando $\Theta(n^3 / \log n)$ procesadores. Por ultimo, la instrucción **repetir** se ejecuta $\lceil \lg n \rceil$ veces, así que el algoritmo completo se puede ejecutar en un tiempo que está en $\Theta(\log^2 n)$ empleando $\Theta(n^3)$ procesadores.

Es sencillo demostrar que el número de procesadores se puede reducir a $\Theta(n^3 / \log n)$ aun manteniendo el orden del tiempo: véase el Problema 11.8. A pesar de ello, el algoritmo sigue sin ser óptimo

11.4.2 Componentes conexos

Sea $G = \langle N, A \rangle$ un grafo no dirigido. Como de costumbre, suponemos que los nodos de G están numerados de 1 a n. Sea un vector L tal que $L[i, j]$= *verdadero* si existe el borde (i, j), y $L[i, j]$= *falso* en caso contrario. Dado que el grafo es no dirigido, $L[i, j] = L[j, i]$ para todo par de nodos (i, j). Deseamos hallar los componentes conexos del grafo G. El problema 9.16 pide al lector hallar un algoritmo secuencial para este problema. Aquí describimos un algoritmo paralelo.

En resumidas cuentas, el algoritmo actúa formando conjuntos disjuntos (véase la Sección 5.9) de nodos de los que sabemos que están conectados, fusionando estos conjuntos con otros más grandes, y así sucesivamente, hasta que finalmente los nodos de cada componente conexo del grafo se encuentran en un único conjunto. Asociamos con cada nodo de G una entrada del vector *conjunto*[1..n]. Si *conjunto*[i]=i, entonces i es a la vez el rótulo de un conjunto y la raíz del árbol correspondiente; si *conjunto*[i] = $j \neq i$, entonces el nodo i está en el conjunto cuyo rótulo es j, pero no la raíz del árbol. Obsérvese que en esta aplicación:

(a) los árboles siempre están «aplanados»; esto es, cada nodo (incluyendo la raíz en sí) apunta directamente a la raíz; y

(b) la etiqueta del conjunto, esto es, el nodo que se encuentra en la raíz del árbol, siempre es el nodo de número más bajo en todo el conjunto.

Comenzaremos por describir tan solo una iteración del algoritmo paralelo que fusiona conjuntos disjuntos. Para ilustrar el funcionamiento del algoritmo, supongamos por ejemplo que tenemos un grafo con 19 nodos, y que de una forma u otra hemos alcanzado la situación que se muestra en la figura 11.4. Aquí los nodos 1, 2

y 3 están en el conjunto marcado como 1, los nodos 4 y 5 están en el conjunto marcado como 4 y así sucesivamente. En términos de nuestra representación, esto significa que *conjunto*[1] = *conjunto*[2] = *conjunto*[3] = 1, *conjunto*[4] = *conjunto*[5] = 4, y así sucesivamente. Se sabe que los nodos de cualquier conjunto dado están conectados, pero también hay algunas conexiones que todavía no se han tenido en cuenta. Estas se indican mediante líneas punteadas en la figura 11.4. De esta forma, el nodo 3, por ejemplo, está conectado con el nodo 7, y el nodo 4 está conectado con el nodo 8; en otras palabras, *L*[3, 7] = *verdadero* , *L*[4, 8]= *verdadero* y así sucesivamente, mientras que por ejemplo *L*[3, 4]= *falso* porque los nodos 3 y 4 no están conectados. Se omiten la conexiones entre nodos del mismo conjunto, por carecer ya de interés.

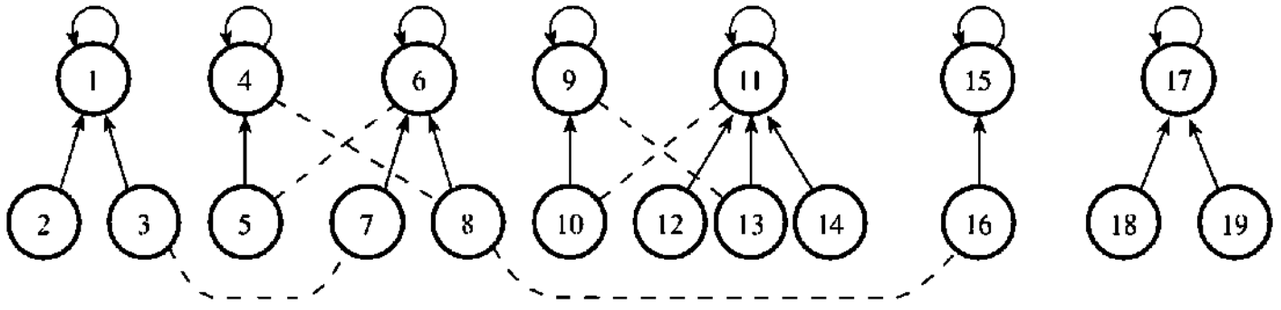


Figura 11.4. Fusión de conjuntos: situación inicial

En la descripción del algoritmo, necesitamos una matriz*S*[1..*n*, 1..*n*], y dos vectores *T*[1..*n*] y*viejoT*[1..*n*]. Ahora se puede especificar en la forma siguiente el primer paso del algoritmo paralelo de fusión:

```
{Paso 1}
  para todo i,j hacer en paralelo
     si L[i, j] y conjunto[i] ≠ conjunto[j]   entonces S[i, j] ← conjunto[j]
                                              sino S[i, j] ← ∞
  para todo i hacer en paralelo T[i] ← min(S[i, 1], S[i, 2],...,S[i, n])
  para todo i hacer en paralelo
     si T[i] = ∞ entonces T[i] ← conjunto[i]
```

Tanto aquí como en el resto de esta sección, las variables *i* y *j* varían entre 1 y *n*.

El efecto de este paso es que para cada nodo *i*, ahora *T*[*i*] apunta a la raíz de un conjunto. Si el nodo *i* está conectado a nodos de otros conjuntos además del suyo, entonces *T*[*i*] apunta a la raíz de uno de estos otros conjuntos: de hecho, al que tenga el número más bajo. Si el nodo *i* no tiene conexiones fuera de su propio conjunto, entonces *T*[*i*] = *conjunto*[*i*].

Aplicando el paso 1 a la situación de la figura 11.4 se obtiene la situación de la figura 11.5, en donde ahora se han omitido las conexiones entre nodos, y las flechas muestran los valores de *T* obtenidos. Todas las flechas apuntan a nodos raíz. De esta forma, por ejemplo, el nodo 1, que no tiene conexiones fuera de su propio

conjunto, tiene $T[1] = conjunto[1] = 1$. El nodo 3, que está conectado con el nodo 8, el cual no está en su propio conjunto, apunta a la raíz de este otro conjunto, a saber, el nodo 6. El nodo 8 proporciona un caso algo más complicado. Está conectado tanto al nodo 4 como al 16, ninguno de los cuales está en el mismo conjunto que el nodo 8. Entonces $T[8]$ debe apuntar o bien a la raíz del conjunto que contiene el nodo 4, o bien a la raíz del conjunto que contiene el nodo 16, esto es, o bien al nodo 4 o bien al nodo 15. Dado que el algoritmo selecciona el número más bajo, obtenemos $T[8] = 4$.

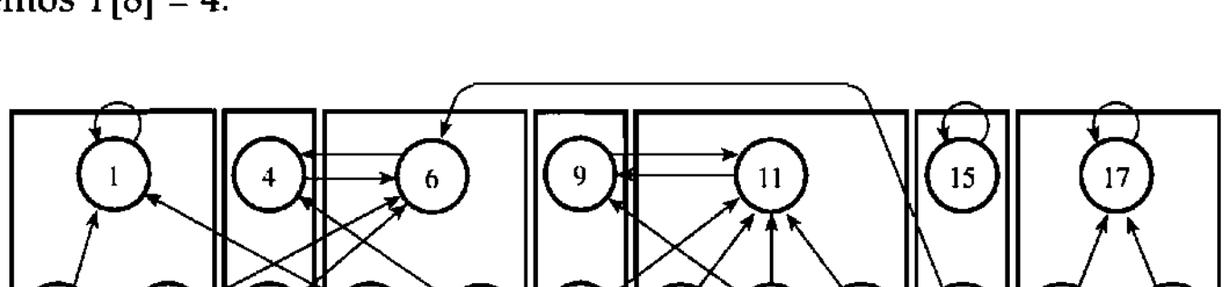


Figura 11.5. Fusión de conjuntos: después del paso 1

El segundo paso del algoritmo de fusión es como sigue.

```
{ Paso 2}
para todo i,j hacer en paralelo
      si conjunto[j] = i y T[j] ≠ i  entonces S[i, j] ← T[j]
                                     sino S[i, j] ← ∞
para todo i hacer en paralelo T[i] ← min(S[i, 1], S[i, 2],...,S[i, n])
para todo i hacer en paralelo
      si T[i] = ∞ entonces T[i] = conjunto[i]
```

Si el nodo i no es la etiqueta de un conjunto, no hay un nodo j que tenga $conjunto[j] = i$, así que este paso se limita a hacer $T[i] \leftarrow conjunto[i]$. Si por otra parte i es un rótulo, entonces el algoritmo examina todos los valores de $T[j]$ para los cuales j sea un nodo del conjunto i, y para los cuales $T[j] \neq i$, esto es, $T[j]$ apunta a otro conjunto diferente. Entonces selecciona el más pequeños de entre ellos. Si no existen tales valores, $T[i] \leftarrow conjunto[i]$, esto es, el rótulo de nodo i se apunta a sí mismo. Esto sólo sucede si ninguno de los nodos del conjunto i está conectado a un nodo de otro conjunto diferente.

Una vez que se aplica el paso 2 a la situación de la figura 11.5, se obtiene la situación ilustrada en la figura 11.6. Las flechas muestran ahora los nuevos valores de T. Todo nodo que no sea un rótulo apunta a la raíz de su propio conjunto, y las flechas entre conjuntos unen solamente nodos raíz.

Considere este grafo dirigido, que llamaremos H: sus nodos son los nodos de G, pero sus aristas están especificadas por los punteros T. Se ha redibuiado en la figura 11.7 para hacer más clara su estructura. Siguiendo el algoritmo, vemos que si uno de los conjuntos iniciales carece de conexión con todos los demás (esto es,

si incluye a todos los nodos de algún componente conexo del grafo G), entonces al cabo de los pasos 1 y 2, los punteros T se limitan a reproducir la estructura inicial del conjunto. Este es el caso del conjunto 17 del ejemplo. Los otros nodos de H forman uno o más componentes conexos, cada uno de los cuales se parece a una pareja de árboles cuyas raíces están unidas en un ciclo. En el ejemplo, un par de árboles tiene como raíces los nodos 1 y 6, y el otro tiene los nodos 9 y 11.

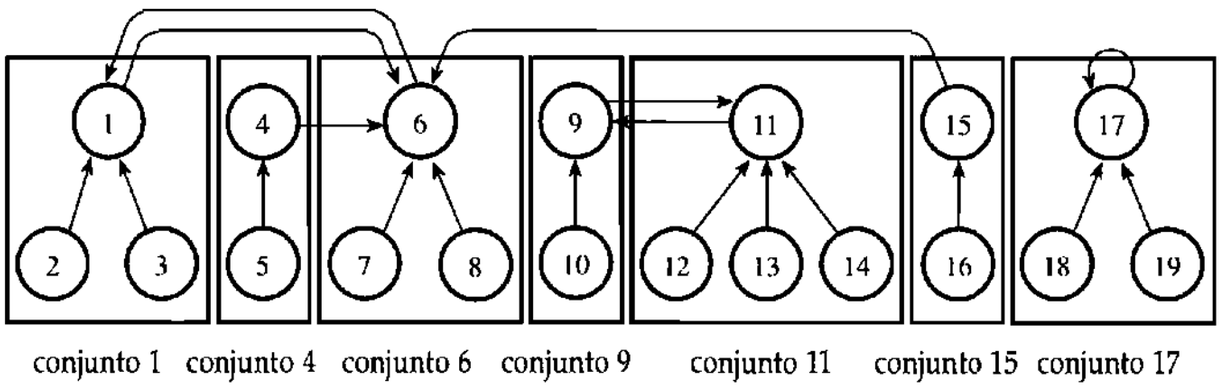


Figura 11.6. Fusión de conjuntos: después del paso 2

Para ver por qué es así, considere un componente de H formado por la fusión de dos o más de los conjuntos originales. Supongamos que a es el rótulodel conjunto de número más bajo de entre los implicados. Dado que la etiqueta del conjunto a es el nodo de número más bajo dentro del conjunto, esto significa que a es de hecho el nodo de número más bajo dentro del componente dado de H. Ahora bien, $T[a] = b$, donde b es la etiqueta de un conjunto distinto de a, puesto que en el paso 2 del algoritmo seleccionamos punteros entre conjuntos distintos siempre que es posible. Además, $T[b] = a$, puesto que si el conjunto a está conectado con el conjunto b, entonces el conjunto b está conectado con el conjunto a (porque el grafo G es no dirigido), y $T[b]$ se ha seleccionado en el paso 2 de manera que sea el menor posible. Entonces $T[a] = b$ y $T[b] = a$, y estos dos nodos forman un ciclo. Los nodos restantes de este componente de H deben estar unidos o bien al nodo a o bien al b mediante una cadena de uno o más punteros T, así que forman dos árboles, uno con a como raíz y otro con b como raíz.

El tercer y último paso del algoritmo utiliza la técnica de duplicación de punteros para aplanar estos árboles dobles, de forma bastante parecida a la Sección 11.2.2. La única sutileza es que si «doblamos» el puntero de un nodo con suficiente frecuencia, entonces estamos seguros de que apuntará a una de las dos raíces, pero no podemos estar seguros de cuál. En el ejemplo, si «doblamos» el puntero del nodo 5, apuntará al nodo 6; todas las duplicaciones subsiguientes dejan esto sin cambios. Si doblamos el puntero del nodo 4, apuntará al nodo 1; una vez más, las duplicaciones subsiguientes no alteran esto. Sin embargo, si las dos raíces son los nodos a y b, hemos visto que antes de toda duplicación de punteros es $T[a] = b$ y $T[b] = a$. Supongamos que guardamos los valores originales de T en el vector *viejoT*. Al cabo de un número suficiente de duplicaciones el puntero del nodo i apunta a una de las dos raíces, y ahora podemos utilizar los valores de *viejoT* para ha-

llar la otra. Comparando, podemos seleccionar la raíz que tenga el número menor. Tal como se indicaba más arriba, se trata del nodo de número más bajo de este componente de H.

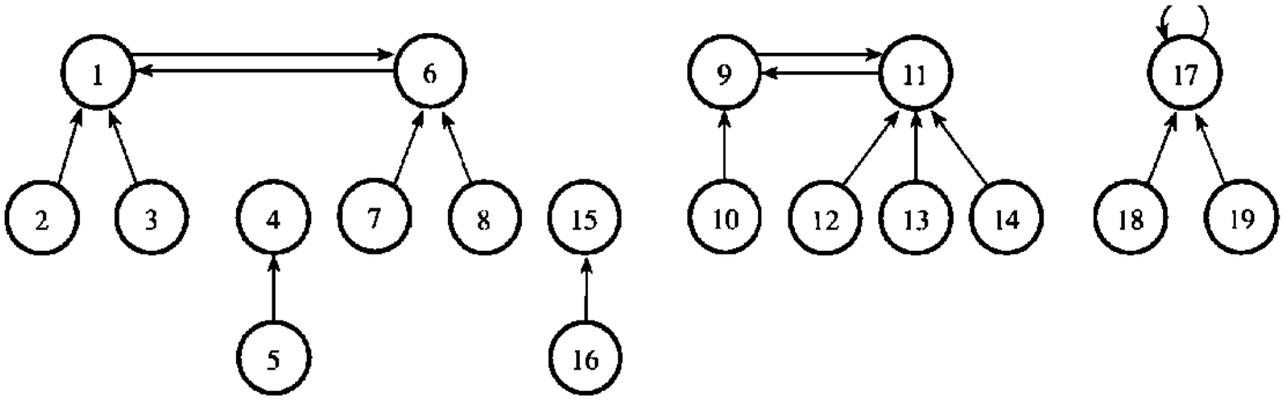


Figura 11.7. Fusión de conjuntos: el grafo H

En un componente de H que tenga una sola raíz, digamos a, al principio es $T[a] = a$, así que la técnica anterior, aun siendo innecesaria, es inofensiva. Dado que todo el grafo G contiene n nodos, ningún componente de H puede componer más de n. Tanto para un árbol sencillo como para uno doble, por tanto, bastan $\lceil \lg n \rceil$ duplicaciones para asegurar que todos los nodos apuntan a una raíz.

Éste es el tercer paso del algoritmo:

```
{ Paso 3}
para todo i hacer en paralelo viejoT[i] ← T[i]
repetir ⌈lg n⌉ veces
    para todo i hacer en paralelo T[i] ← T[T[i]]
    para todo i hacer en paralelo conjunto[i] ← min(T[i],viejoT[T[i]])
```

Supongamos que algún componente conexo de G estuviera representado inicialmente por más de un conjunto disjunto. Tras la ejecución de los pasos 1, 2, y 3 algunos de ellos se habrán fusionado en otros más grandes. De hecho, todo conjunto que represente solamente una parte de un componente conexo de G debe de estar enlazado al menos con otro conjunto distinto que represente una parte diferente del mismo componente. Por tanto tales conjuntos deben fusionarse de dos en dos en el caso peor. En otras palabras, en el caso peor el número de conjuntos disjuntos que representan el mismo componente conexo de G se divide por dos mediante la aplicación de los pasos 1, 2 y 3.

El algoritmo paralelo completo para hallar los componentes conexos de un grafo no dirigido G con n nodos se puede especificar por tanto en la forma siguiente. En primer lugar, cada nodo de G se pone en un conjunto disjunto diferente. A continuación se iteran los pasos 1, 2 y 3 durante $\lceil \lg n \rceil$ veces. Dado que el número de conjuntos disjuntos que representan a un mismo componente conexo de G se divide al menos por dos en cada iteración, es seguro que esto efectuará todas las posibles fusiones, aun cuando G tuviera un único componente conexo. Cuando se

detenga el algoritmo, los conjuntos disjuntos restantes representarán distintos componentes conexos de G. El algoritmo es el siguiente:

```
procedimiento compcons(L[1..n, 1..n]): matriz [1..n]
{L es la matriz booleana de adyacencia de un grafo G
no dirigido. El algoritmo proporciona un vector que
representa una estructura de conjuntos disjuntos, en
donde cada conjunto disjunto corresponde a un
componente conexo de G}
   matriz conjunto[1..n], S[1..n, 1..n], T[1..n], viejoT[1..n]
   para todo i hacer en paralelo conjunto[i] ← i
   repetir ⌈lg n⌉ veces
      Paso 1
      Paso 2
      Paso 3
   devolver conjunto
```

Análisis del algoritmo

El análisis del algoritmo es sencillo. En primer lugar, se iteran los tres pasos $\lceil \lg n \rceil$ veces, y dado que cada ejecución del paso 3 da lugar a que se ejecute $\lceil \lg n \rceil$ veces la instrucción de duplicación de punteros, el tiempo de ejecución del algoritmo está en $\Omega(\log^2 n)$, independientemente del número de procesadores que estén disponibles.

Ahora se puede ejecutar la primera instrucción **para** del paso 1 en un tiempo constante, con n^2 procesadores. En la segunda instrucción **para** de este paso, el mínimo de n valores de un vector se puede hallar en un tiempo que está en $\Theta(\log n)$ empleando $\Theta(n^2)$ procesadores. La tercera instrucción **para** se puede ejecutar en tiempo constante empleando n procesadores. En consecuencia, el total del paso 1 se puede ejecutar en un tiempo que está en $\Theta(\log n)$ empleando $\Theta(n^2)$ procesadores. Es fácil comprobar que no hay conflictos de escritura durante este paso. El análisis del paso 2 es exactamente paralelo al del caso 1.

En lo tocante al caso 3, las instrucciones primera y última se pueden ejecutar en tiempo constante con n procesadores, mientras que la instrucción **repetir**, tal como se vio, requiere un tiempo que está en $\Theta(\log n)$ y se puede ejecutar con n procesadores. Una vez más, no hay conflictos de escritura.

Reuniendo todos estos hechos, vemos que se puede ejecutar el algoritmo *compcons* en un tiempo que está en $\Theta(\log^2 n)$ empleando $\Theta(n^2)$ procesadores. El trabajo efectuado se encuentra en $\Theta(n^2 \log^2 n)$, que no es óptimo. De hecho, se puede mejorar con facilidad. Los puntos críticos del algoritmo que determinan el número de procesadores necesarios son los cálculos de n mínimos en los pasos 1 y 2. Empleando el resultado del problema 11.2, vemos que lo cierto es que se puede calcular un mínimo en un tiempo que está en $\Theta(\log n)$ empleando un

número de procesadores que está en $\Theta(n/\log n)$; entonces los n mínimos se pueden calcular en el mismo tiempo empleando $\Theta(n^2/\log n)$ procesadores. Con esta mejora, el algoritmo *compcons* se puede ejecutar en un tiempo que está en $\Theta(\log^2 n)$ empleando $\Theta(n^2/\log n)$ procesadores. Sin embargo, esto sigue sin ser óptimo.

El algoritmo puede mejorarse todavía más aprovechando el hecho consistente en que a medida que progresan los cálculos va disminuyendo el número de conjuntos disjuntos que hay que manejar, y se necesitan menos procesadores. Sin embargo, esto da lugar a un algoritmo paralelo para el problema de los componentes conexos que sigue requiriendo un tiempo que está en $\Theta(\log^2 n)$, pero ahora se emplean solamente $\Theta(n^2/\log^2 n)$ procesadores. El trabajo efectuado por este algoritmo mejorado está en $\Theta(n^2)$. Dado que ningún algoritmo secuencial que utilice el vector de adyacencia del grafo G puede hacerlo mejor, el algoritmo paralelo mejorado es óptimo.

11.5 EVALUACIÓN DE EXPRESIONES EN PARALELO

Existe una notable cantidad de literatura acerca de este tema, que no pretendemos resumir aquí. Daremos tan solo un ejemplo de la forma de computar expresiones sencillas en paralelo. El ejemplo seleccionado es fácil de explicar; sin embargo, la técnica de resolución resulta gratamente novedosa.

Supongamos que se está buscando un algoritmo paralelo para evaluar expresiones aritméticas sencillas, cuyos operandos son constantes, y que solamente involucran a los cuatro operadores $+, -, \times$ y $/$. A lo largo de esta sección supondremos que estas cuatro operaciones aritméticas son elementales, esto es, que se pueden calcular en tiempo constante. Por sencillez, suponemos que la expresión que hay que evaluar se encuentra en la forma de un *árbol de expresión*: se trata de un árbol binario en el que cada nodo interno representa uno de los cuatro operadores disponibles, y cada hoja denota un operando. La figura 11.8 muestra uno de estos árboles, correspondiente a la expresión

$$\big((7-(21/3))\times 3\big)+\big((9\times(10-8))+6\big)$$

Además, suponemos que las hojas del árbol están numeradas de izquierda a derecha siguiendo el borde inferior del árbol, también según se ilustra en la figura 11.8. La x siguiendo el borde inferior que hay a la derecha de los nodos internos de esta figura se explicará más adelante. Si el árbol tiene n hojas (operandos) entonces desde luego tiene $n-1$ nodos (operadores). En el ejemplo, $n = 8$.

La forma evidente de evaluar paralelamente uno de estos árboles de expresión consiste en asignar un procesador a cada nodo interno, y utilizar un algoritmo de la forma siguiente:

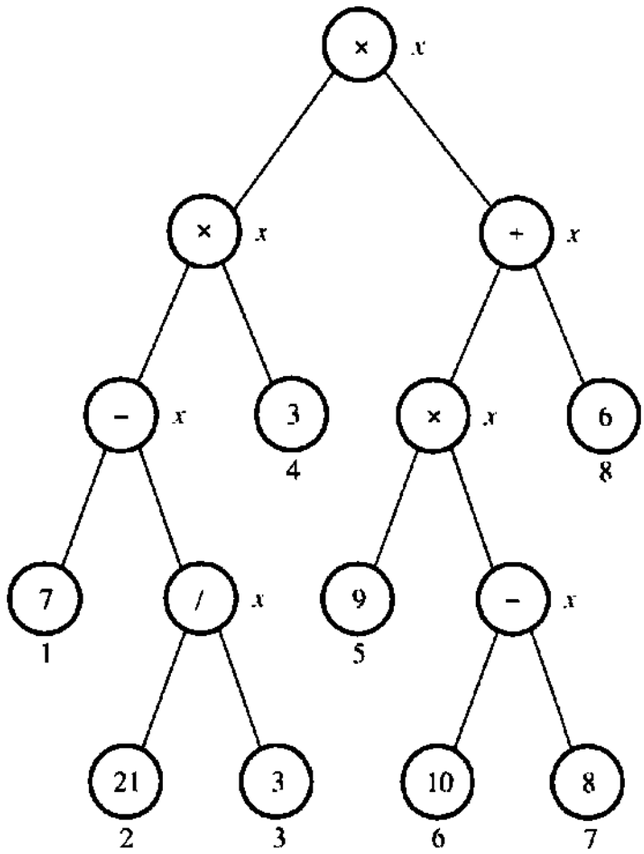


Figura 11.8. Un árbol de expresión binaria.

repetir
  para cada nodo interno *i* **hacer en paralelo**
    si los valores de los hijos de *i* son conocidos **entonces**
      calcular el valor de *i*
      eliminar del árbol los hijos de *i*
hasta que sólo quede un nodo

El número de iteraciones necesarias es igual a la altura del árbol de expresión. En el caso peor, sin embargo, un árbol binario con n hojas puede tener una altura $n-1$, y este algoritmo sencillo ¡no produce paralelismo alguno! Sólo un procesador efectúa un trabajo útil en cada iteración, y para eso el cálculo podría hacerse en una máquina secuencial; véase el problema 11.10.

Para acelerar las cosas, los procesadores deben de hacer algo útil incluso antes de que se hayan evaluado sus dos hijos. Aquí buscamos algo que pueda hacer un procesador cuando se conoce el valor de al menos uno de sus hijos. Con este objeto, asociamos una función $f(x)$ a cada nodo interno del árbol. Inicialmente, todos los nodos internos están asociados con la función x, según se ilustra en la figura 11.8. El significado de estas funciones es que cuando un procesador de algún nodo ha calculado un valor x, el valor que transmite hacia arriba por el árbol no es x sino $f(x)$. Considere por ejemplo el fragmento de árbol que se muestra en la Figura 11.9a. Aquí el procesador asignado al nodo interno A recibe un valor de su hijo izquierdo, y un valor del derecho, los multiplica para obtener una respuesta x, y después transmite el valor $f(x)$ hacia arriba, al nodo B. A su vez, el procesador asociado al nodo B recibe este valor de su hijo izquierdo A y el valor 9 de su hijo derecho C (que es una hoja, y corresponde

a un operando constante), los suma para obtener una respuesta x y transmite entonces el valor $g(x)$ por el árbol hasta su predecesor. Y así sucesivamente.

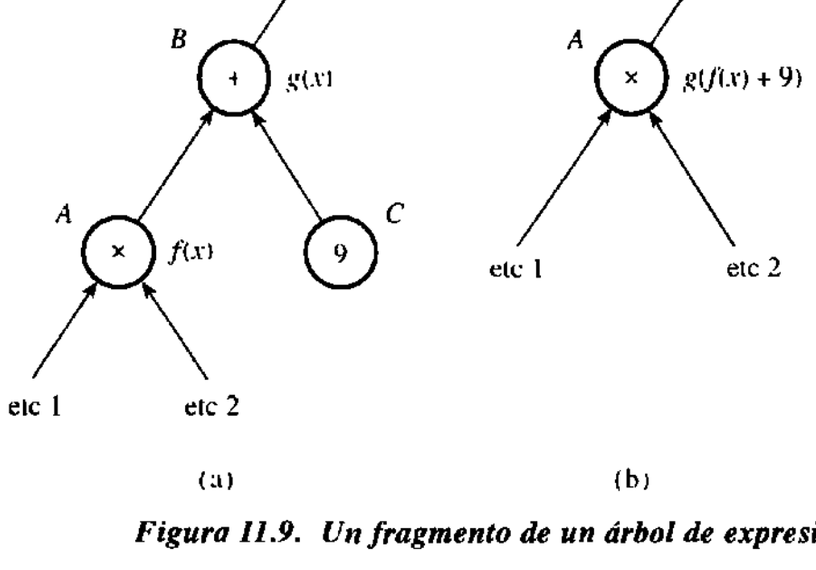


Figura 11.9. Un fragmento de un árbol de expresión

Incluso antes de recibir un valor del nodo A, sin embargo, el procesador del nodo B puede efectuar un trabajo útil, reconstruyendo el árbol de expresión. Dado que se conoce el valor del hijo derecho del nodo B, puede modificar la función almacenada en el nodo A y eliminar el nodo B y su hijo (el derecho, en este caso) del árbol. El resultado se muestra en la figura 11.9b. Si cuando calcule el valor de x el procesador del nodo A transmite directamente $g(f(x) + 9)$ al predecesor de B, soslayando a A y a su hijo derecho, entonces el resultado obtenido en la raíz del árbol no varía.

Nada hay de especial acerca del operador + del nodo B, ni tampoco acerca del valor 9 que le dimos a su hijo derecho C. En general, si el nodo B contiene el operador $\circ$, en donde $\circ$ es cualquiera de entre +, −, × o /,y si el valor de su hijo derecho es cualquier constante k, entonces B puede sustituir la función de A por $g(f(x)\circ k)$ y eliminarse a sí mismo y a su hijo derecho del árbol de expresión. Si la constante k es el hijo izquierdo de B y el nodo A es el hijo derecho de B, entonces B debe de sustituir la función de A por $g(k\circ f(x))$. Esto es importante porque los operadores − y / no son conmutativos. Unos ajustes menores resuelven los casos en que A es una hoja (se toma como $f(x)$ la función constante que proporciona el valor del operando que hay en A) o bien cuando B es la raíz del árbol (el valor final del árbol es el valor que normalmente se elevaría hasta el nivel siguiente).

La operación descrita más arriba se llama *corte*. En el ejemplo, la hoja C y su predecesor B se han cortado del árbol. Es importante para lo que sigue ver que un corte se puede efectuar en tiempo constante. La manipulación de punteros del árbol se puede efectuar ciertamente en tiempo constante: sólo tres punteros están implicados en la operación. Resulta menos evidente que la función asociada a un nodo se pueda actualizar rápidamente. Por ejemplo, si en el ejemplo los nodos A y B se han creado ambos mediante una serie precedente de operaciones de corte, las funciones $f(x)$ y $g(x)$ podrían ya ser —o eso parece— bastante complejas, así que al sustituir f en g para obtener $g(f(x)\circ k)$ no sería trivial.

Afortunadamente, si los únicos operadores admitidos son +, −, × y /, las funciones que se pueden obtener de esta forma tienen todas ellas la forma $(ax + b)(cx + d)$. Para ver esto, observe primero que la función inicial de todos los nodos internos, a saber, x, se puede representar mediante $a = d = 1$ y $b = c = 0$. Si $f(x) = (a_1x + b_1) / (c_1x + d_1)$, y $g(x) = (a_2x + b_2) / (c_2x + d_2)$, entonces $g(f(x)) = (a_3x + b_3) / (c_3x + d_3)$, en donde es $a_3 = a_1a_2 + b_2c_1$ y existen otras expresiones igualmente sencillas para b_3, c_3 y d_3; véase el problema 11.11. Por tanto, para representar cualquier función asociada con un nodo interno sólo hay que mantener las cuatro constantes correspondientes, a, b, c y d. Siempre que las constantes no sean demasiado grandes, la representación de cualquier composición funcional de la forma $g(f(x)°k)$ o $g(k°f(x))$ se puede computar en tiempo constante siempre que sea necesario. Además, cuando el valor de x esté disponible, $f(x)$ también se puede calcular en tiempo constante. (La salvedad es necesaria porque en una expresión complicada las constantes pueden crecer exponencialmente. Sin embargo, aquí no consideraremos esta posibilidad.)

La última consideración antes de enunciar el algoritmo para evaluar expresiones es que en una operación de corte sólo están implicados tres nodos. En la Figura 11.9, estos nodos son A, B y C. Los punteros, valores y funciones asociadas se leen y se modifican para estos tres nodos y ninguno más. Por tanto es posible ejecutar una operación de corte en alguna otra parte del árbol, en paralelo con la operación que afecta a A, B y C, siempre y cuando ninguno de ellos esté implicado en la segunda operación. Además, sean A, B y C los nodos implicados en una operación de corte, y A', B' y C' los nodos implicados en otra, siendo C y C' las dos hojas implicadas. Recuerde que las hojas del árbol de expresión están numeradas por orden en torno al árbol. Entonces es condición suficiente para que las operaciones no interfieran entre sí —esto es, para que los conjuntos $\{A, B, C\}$ y $\{A', B', C'\}$ sean disjuntos— el que C y C' sean hojas no consecutivas y que sean ambos hijos izquierdos o hijos derechos. El problema 11.12 pide al lector que demuestre esta afirmación.

El algoritmo paralelo completo para evaluar expresiones simples se puede enunciar ahora en la forma siguiente. Suponemos que se reserva un procesador para cada nodo interno del árbol de expresión:

```
función evalp(T: árbol de expresión)
{Evalúa un árbol de expresión con n hojas y n−1
nodos internos. Las hojas están numeradas inicialmente
por orden siguiendo la base del árbol}
   para cada nodo interno de T hacer en paralelo
      iniciar la función f(x) con el valor x
   repetir ⌈lg n⌉ veces
      para cada nodo interno de T hacer en paralelo
         si el hijo izquierdo es una hoja impar, cortarlo
         si el hijo derecho es una hoja impar, cortarlo
         si alguno de los hijos es ahora una hoja, cambiar su número
   devolver el valor de la hoja restante
```

En primer lugar, se cortan todas las hojas impares que sean hijos izquierdos. Por los comentarios anteriores, se puede hacer esto en paralelo sin que las operaciones de corte interfieran unas con otras. A continuación se cortan todas las hojas impares que sean hijos derechos. Una vez más, esto se puede hacer en paralelo. Dado que todas las hojas impares son o bien un hijo izquierdo o bien un hijo derecho, se habrán eliminado todas al llegar este momento, y solamente quedan las $\lfloor n/2 \rfloor$ hojas pares. Éstas se renumeran dividiendo su número por dos, y estamos preparados para la próxima iteración. Dado que cada iteración elimina al menos la mitad de las hojas del árbol de expresión, al cabo de $\lceil \lg n \rceil$ iteraciones sólo queda una hoja. El valor de esta hoja es el valor de la expresión.

La figura 10.10 ilustra una iteración de este proceso cuando se le aplica al árbol de expresión de la figura 11.8. La primera mitad de la figura muestra el estado del árbol después de haber cortado las hojas izquierdas de número impar, y la segunda mitad muestra el estado después de haber recortado las hojas derechas de número impar. Se aprecia fácilmente que si ahora se dividen por dos los números de las hojas, el árbol quedará preparado para la próxima iteración. El lector puede verificar que la segunda iteración reduce el árbol a tres nodos (un nodo interno y dos hojas), mientras que la tercera lo reduce a un único nodo que contiene el valor de la expresión original, a saber, 24.

Dado que un corte se puede efectuar en tiempo constante, es fácil ver que el algoritmo descrito más arriba requiere un tiempo que está en $\Theta(\log n)$, empleando $\Theta(n)$ procesadores. El trabajo efectuado está en $\Theta(n \log n)$, así que el algoritmo no es óptimo. Tal como en los ejemplos anteriores, al cabo de una iteración sólo sigue siendo útil la mitad de los procesadores; después de dos iteraciones sólo actúa la cuarta parte, y así sucesivamente. A costa de una mayor complejidad, podemos

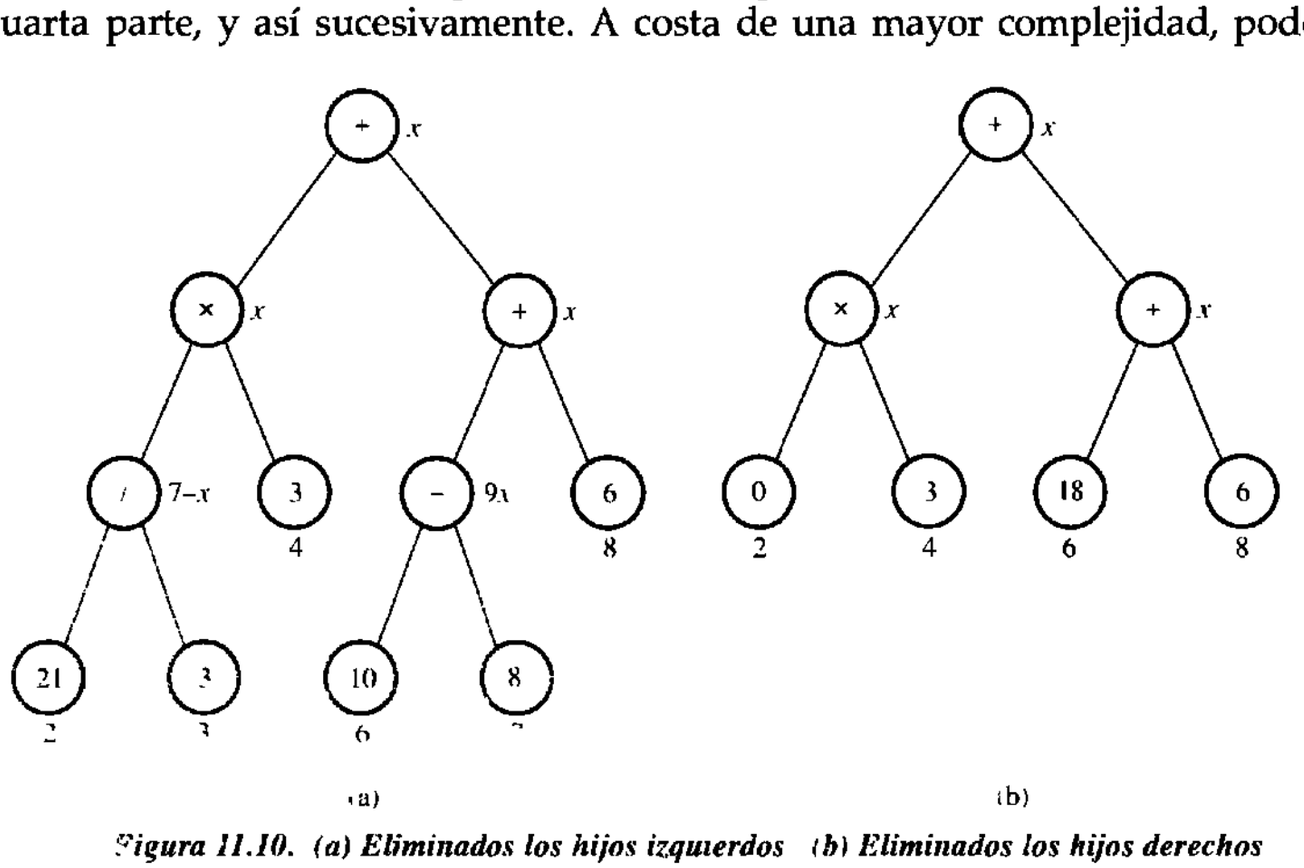


Figura 11.10. (a) Eliminados los hijos izquierdos (b) Eliminados los hijos derechos

aprovechar esto para reducir el número de procesadores necesario hasta $\Theta(n/\log n)$ sin que el tiempo requerido vaya más allá de $\Theta(\log n)$. El algoritmo mejorado funciona en $\Theta(n)$, y por tanto es óptimo.

La forma de la entrada necesaria para el algoritmo anterior (un árbol de expresión con las hojas numeradas de izquierda a derecha) puede parecer poco habitual. Aunque aquí omitiremos los detalles, observamos que si la expresión que hay que calcular no tiene la forma requerida, sino que está almacenada en forma de una cadena (esto es, un vector de caracteres) entonces el árbol de expresión numerado se puede obtener en un tiempo que está en $\Theta(\log n)$ empleando $\Theta(n/\log n)$ procesadores: son exactamente las mismos órdenes que para el algoritmo de evaluación. Por tanto, la conversión de la entrada a la forma requerida no constituye una pérdida de rendimiento.

11.6 REDES DE ORDENACIÓN EN PARALELO

Antes de utilizar el problema de la ordenación en paralelo empleando un *p-ram* CREW, haremos una momentánea digresión para examinar una interesante clase de redes que son capaces de ordenar sus entradas.

Comenzaremos por definir un *comparador*. Se trata de un circuito con dos entradas y dos salidas, que representaremos tal como en la figura 11.11. Por convención, suponemos que las entradas están en el lado izquierdo y las salidas en el derecho. Si como se muestra en la figura las dos entradas son x_1 y x_2, y las dos salidas son y_1 e y_2, entonces es $y_1 = \min(x_1, x_2)$ e $y_2 = \max(x_1, x_2)$. En otras palabras, la entrada mayor baja a la salida inferior, mientras que la entrada menor sube a la salida superior. En lo que sigue, será cómodo suponer que las entradas se intercambian cuando $x_1 \leq x_2$. De esta manera, es como si las entradas iguales se intercambiaran, aunque por supuesto esto no tiene ningún efecto visible. Claramente, un solo comparador es capaz de ordenar dos entradas. Supondremos que esta operación de comparación se puede efectuar en un tiempo constante. También suponemos que puede funcionar en paralelo cualquier cantidad de comparadores, siempre que sus entradas y salidas sean disjuntas. En la práctica, resulta fácil construir comparadores que satisfagan estos requisitos

Más generalmente, deseamos diseñar redes que admitan n entradas. Si se aplica cualquier vector $(x_1, x_2, \ldots, x_n)$ a las entradas de una de estas redes, entonces el vector de salida $(y_1, y_2, \ldots, y_n)$ será una permutación de las entradas tal que $(y_1 \leq y_2 \leq \ldots \leq y_n)$. Si denotamos mediante S_n una de estas redes, entonces S_1 no requiere ningún com-

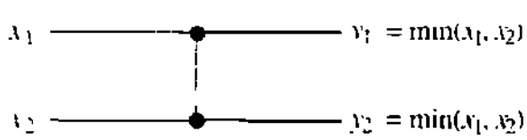


Figura 11.11. Un comparador

parador, mientras que S_2, como acabamos de ver, es un único comparador. Una forma de construir redes cada vez más grandes consiste en diseñar S_{n+1} en términos de S_n, de tal manera que a partir de S_1 o de S_2 se pueden construir todas las redes que se deseen. Hay por los menos dos maneras evidentes de hacerlo, que se ilustran en la figura 11.12. Ambas redes poseen n+1 entradas y n+1 salidas. La red de la izquierda corresponde a ordenar por selección: el mayor elemento cae al fondo, y entonces utilizamos S_n para ordenar los n valores restantes. La red de la derecha corresponde a la ordenación por inserción: utilizamos S_n para ordenar las n primeras entradas, y entonces se inserta la (n+1)-ésima entrada en el lugar adecuado. Curiosamente, cuando comparamos las redes obtenidas de esta manera, resultan ser la misma. La figura 11.13 ilustra la red S_5 que se obtiene tanto si se usa la inserción como empleando la selección.

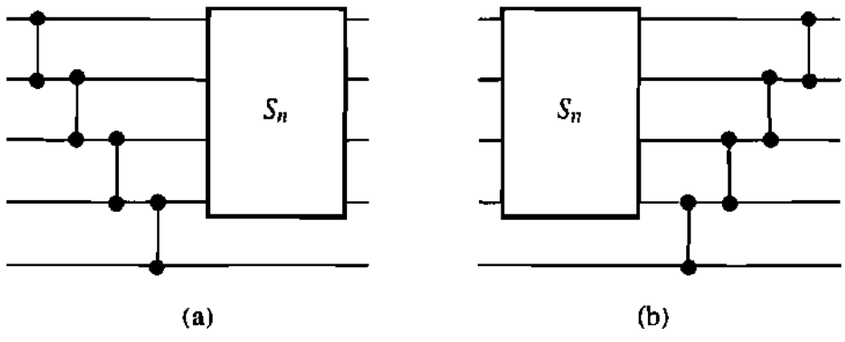


Figura 11.12. Redes de ordenación por selección e inserción

Existen dos medidas útiles de la calidad de nuestras redes. En primer lugar, podemos contar simplemente el número de comparadores necesarios para construir S_n. Esto se llama *tamaño* de la red. En el ejemplo, S_5 contiene 10 comparadores, y es evidente que en general S_n contendrá $\sum_{i=1}^{n-1} = n(n-1)/2$ comparadores. La segunda medida de interés es el tiempo que necesita la red para ordenar sus entradas. Suponíamos que un comparador requiere un tiempo constante para funcionar, pero desde luego no puede activarse mientras sus salidas no estén preparadas. Definimos la *profundidad* de una red como el máximo número de comparadores a través de los cuales tiene que pasar una entrada antes de que llegue a una salida. La profundidad de la red que aparece en la figura 11.13 es 7. Los comparadores de la misma línea vertical pueden funcionar a la vez, mientras que las líneas sucesivas tienen que ejecutarse de izquierda a derecha. En este caso, la entrada 2 pasa por 7 comparadores. En general, resulta sencillo ver que una red S_n diseñada de esta manera tiene una profundidad $2n-3$ cuando $n \geq 2$. Si cada fase requiere un tiempo constante, el tiempo necesario para ordenar n elementos empleando este tipo de red de ordenación paralela es proporcional a la profundidad de la red, y por tanto está en $\Theta(n)$.

En las secciones siguientes veremos la forma de mejorar este diseño.

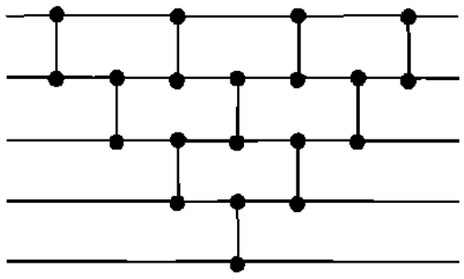


Figura 11.13. Una red para ordenar cinco entradas

11.6.1 El principio cero-uno

Para demostrar que nuestras redes de ordenación funcionan como deberían hacerlo, emplearemos la siguiente proposición, ligeramente sorprendente, y conocida con el nombre de *principio cero-uno*.

Lema 11.6.1. *Una red de ordenación con n entradas ordena correctamente cualquier conjunto de valores dado en sus entradas si y solo si ordena correctamente todos los 2^n vectores de entrada que constan únicamente de ceros y unos.*

Demostración La parte «sólo si» de la proposición es obvia. Para demostrar la parte "si", sea f: $\mathbb{R} \rightarrow \mathbb{R}$ cualquier función no decreciente, de modo que $f(x) \leq f(y)$ siempre que $x \leq y$. Supongamos que la red de ordenación considerada ordena correctamente todos los 2^n vectores de entrada que constan solamente de ceros y unos, pero que existe algún vector de entradas $(x_1, x_2, \ldots, x_n)$ que se ordena de forma incorrecta. Obsérvese que incluso una red de ordenación incorrecta produce una permutación de sus entradas. Sea $(y_1, y_2, \ldots, y_n)$ el vector de salida (incorrecto) para este conjunto de entradas, y sea y_i cualquier elemento de este vector tal que $y_i > y_{i+q}$.Este elemento tiene que existir necesariamente, puesto que el vector está mal ordenado.

Consideremos ahora lo que sucedería si en lugar de $(x_1, x_2, \ldots, x_n)$ aplicásemos a la red el vector de entrada $(f(x_1), f(x_2), \ldots, f(x_n))$. Dado que f es no decreciente, $f(x_i) \leq f(x_j)$ siempre que sea $x_i \leq x_j$. De esta manera, los valores de $f(x_i)$ se propagan por la red de ordenación exactamente de igual manera que los valores de x_i: del mismo modo que dos valores de x se intercambiaban, se intercambian ahora los dos valores correspondientes de $f(x)$ (esta es la razón por la cual exigíamos que los comparadores intercambiasen dos valores iguales.) El vector de salida de la red de ordenación será $(f(y_1), f(y_2), \ldots, f(y_n))$, puesto que la red va a efectuar exactamente la misma permutación de sus entradas que en el caso anterior.

Por último, sea f la función definida en la forma siguiente: $f(x) = 0$ cuando $x < y_i$, y $f(x) = 1$ en caso contrario. Ahora el vector de entradas $(f(x_1), f(x_2), \ldots, f(x_n))$ es un vector que consta únicamente de ceros y unos. En el vector de salida $f(y_i) = 1$ y $f(y_{i+1}) = 0$, porque $y_{i+1} < y_i$. El vector de salida, por tanto, estará incorrectamente ordenado, lo cual contradice nuestra suposición de que la red ordena correctamente todos los vectores de en-

trada que contengan únicamente unos y ceros. Se sigue que no puede existir ninguno de estos vectores $(x_1, x_2, \ldots, x_n)$: la red ordena correctamente *cualquier* vector de entrada, y nuestra demostración del principio cero-uno está terminada.

11.6.2 Redes de fusión en paralelo

Necesitamos una herramienta más antes de volver a las redes de ordenación. Para todo entero positivo n, una *red de fusión* F_n es una red formada por comparadores, con dos grupos de n entradas y un solo grupo de $2n$ salidas. Siempre y cuando cada uno de los dos grupos de entradas esté ya ordenado, cada entrada aparecerá en una de las salidas, y las salidas también estarán ordenadas (compárese esto con la descripción de la fusión en la Sección 7.4.1). Por ejemplo, la figura 11.14 muestra una red F_4, que ilustra la forma en que las entradas se transmiten a las salidas.

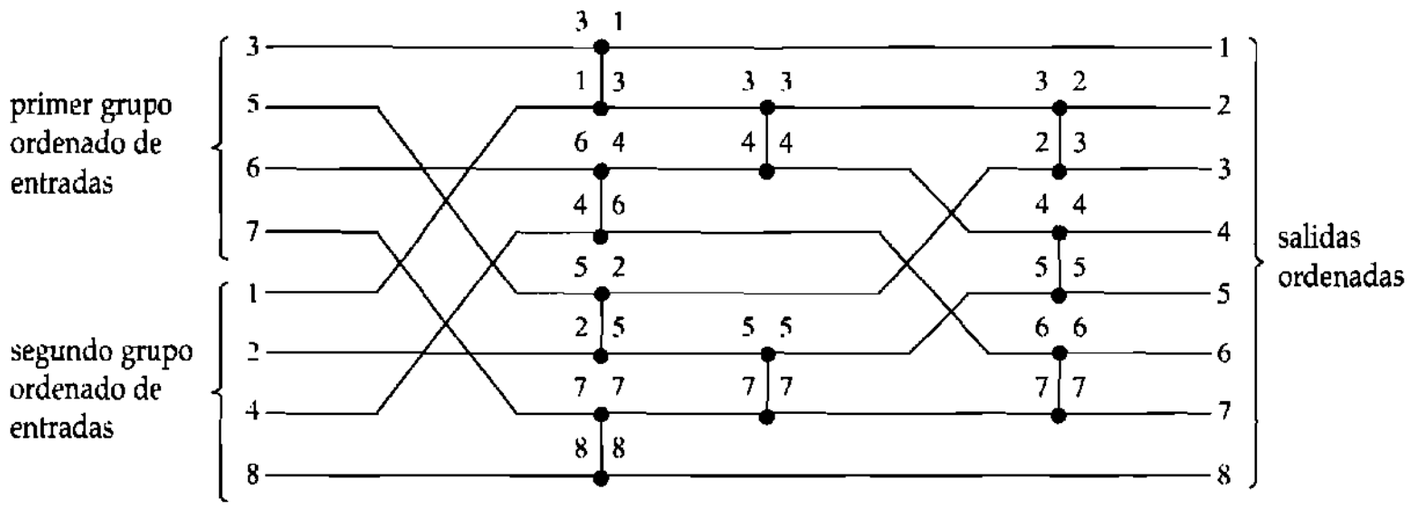


Figura 11.14. Una red de fusión

Por sencillez, supondremos a partir de ahora que n es una potencia de 2. Un solo comparador puede servir como F_1. Partiendo de esta base creamos redes de fusión F_2, F_4 y así sucesivamente, diseñando siempre a F_{2n} en términos de F_n. Este es otro ejemplo de la técnica de divide y vencerás que se describe en el Capítulo 7. La figura 11.15 muestra la forma de hacerlo. Supongamos que los dos grupos de entradas son $(w_1, w_2, \ldots, w_n)$ y $(x_1, x_2, \ldots, x_n)$, y que las salidas son $(y_1, y_2, \ldots, y_{2n})$. Fusionamos las entradas impares $(w_1, w_3, \ldots, w_{n-1})$ del primer grupo con las entradas impares $(x_1, x_3, \ldots, x_{n-1})$ del segundo grupo empleando una red de fusión F_n, y fusionamos las entradas pares $(w_2, w_4, \ldots, w_n)$ del primer grupo con las entradas pares $(x_2, x_4, \ldots, x_n)$ del segundo grupo empleando otra red. Llamaremos a las salidas de estas dos fusiones $(v_1, v_2, \ldots, v_{2n})$, numerando las salidas de la fusión impar antes que las salidas de la fusión par. Ahora se permutan las salidas v_i de tal manera que, desde arriba hasta abajo, se encuentren en el orden $(v_1, v_{n+1}, v_2, v_{n+2}, \ldots, v_n, v_{2n})$. Esta permutación es lo que se denomina *barajado perfecto*: si cortamos un mazo de $2n$ cartas exactamente por la mitad, y las mezclamos de tal modo que vaya cayendo alternativamente una carta de cada mitad, el orden que se obtiene es éste. Por último, se ins-

talan comparadores entre lo que ahora son las salidas 2 y 3, 4 y 5, $2n-2$ y $2n-1$. La salida de la derecha es el vector ordenado que se deseaba, $(y_1, y_2, \ldots, y_{2n})$.

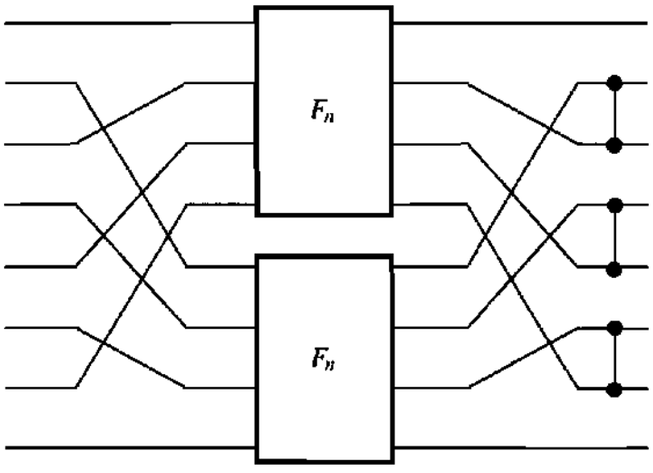


Figura 11.15. Diseño de F_{2n} en términos de F_n.

Un argumento exactamente análogo al dado anteriormente muestra que el principio cero-uno también es válido para redes de fusión. Por tanto, para demostrar que funciona la red propuesta, mostramos primero que funciona cuando las entradas w y x constan solamente de unos y ceros, y después invocamos el principio cero-uno para concluir que también funciona para entradas cualesquiera. El argumento es por inducción matemática. Como base para la inducción, es evidente que un único comparador, F_1, funciona correctamente. Como paso de inducción, supongamos demostrado que todas las redes $F_1, F_2, \ldots, F_n$ funcionan correctamente. Considérese lo que hace nuestro F_{2n} propuesto cuando el vector de entrada w consta de r ceros seguidos por $n-r$ unos, y x consta de s ceros seguidos por $n-s$ unos. (Recuerde que ambos grupos de entradas deben estar ya ordenados.) Dado que por la hipótesis de inducción las redes F_n funcionan correctamente, la salida $(v_1, v_2, \ldots, v_n)$ de la red superior de fusión consta de $\lceil r/2 \rceil + \lceil s/2 \rceil$ ceros seguidos por el número adecuado de unos, mientras que la salida $(v_{n+1}, v_{n+2}, \ldots, v_{2n})$ de la red de fusión inferior consta de $\lfloor r/2 \rfloor + \lfloor s/2 \rfloor$ ceros seguidos por unos. Si tanto r como s son pares, entonces después de barajar las salidas de las dos redes de fusión, las $2n$ líneas que van desde arriba hasta abajo contienen una vez más $r+s$ ceros seguidos por unos; tal como antes, la columna final de comparadores resulta innecesaria. Si tanto r como s son impares, sin embargo, entonces después de barajar las salidas de las dos redes de fusión, los valores de las líneas que van desde arriba hacia abajo son $r+s-1$ ceros, un uno, un cero, y después unos. Esta vez el comparador entre las líneas $r+s$ y $r+s+1$ es necesario para finalizar la ordenación. El F_{2n} propuesto ordena por tanto correctamente todas las entradas que consten únicamente de ceros y unos; por el principio cero-uno, ordenará correctamente entradas cualesquiera.

La figura 11.14 se obtuvo de esta manera, salvo que se ha retocado la red para eliminar algunos cruces de líneas redundantes.

Para calcular el tamaño $s(n)$ de la red F_n obtenida empleando esta construcción, utilizaremos la recurrencia

$$s(2n) = 2s(n) + n/2-1$$

que es inmediata a partir de la Figura 11.15. La condición inicial es $s(1) = 1$. Empleando los métodos de la Sección 4.7 se sigue fácilmente que $s(n) = 1 + n \lg n$. Resulta igualmente fácil mostrar que la profundidad de la red F_n es $1 + \lg n$.

11.6.3 Redes de ordenación mejoradas

Ahora podemos utilizar la técnica de divide y vencerás para diseñar redes de ordenación S_n que sean una mejora notable con respecto a las vistas al principio de la Sección 11.6. La figura 11.16 muestra como hacerlo: se utilizan dos redes S_n para ordenar por separado las n primeras entradas, y las n últimas, y después se utiliza un circuito de fusión F_n para completar la ordenación. No es necesario demostrar que la red S_{2n} funciona correctamente si las dos redes menores S_n funcionan también correctamente. El enfoque de divide y vencerás se detiene en S_2, que consta de un solo comparador.

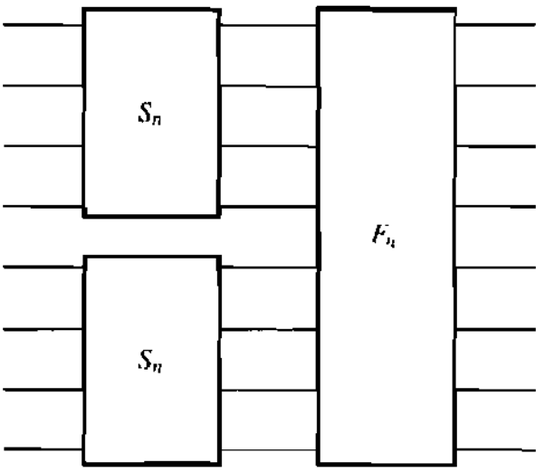


Figura 11.16. Una red de ordenación mejorada.

Resulta fácil mostrar que las redes de ordenación obtenidas de esta manera emplean un número de comparadores que está en $\Theta(n \log^2 n)$ y que el tiempo que necesitan para ordenar sus entradas está en $\Theta(\log^2 n)$. Por razones evidentes, las redes descritas más arriba se denominan redes par-impar de ordenación o fusión. Fueron descubiertas por Batcher en 1964. Estas redes de ordenación no son óptimas. Anticipándonos un poco, veremos en el capítulo siguiente que *cualquier* algoritmo que ordena n elementos mediante comparaciones entre ellos debe hacer al menos $\lceil lg\, n! \rceil$ en el caso peor. Por tanto, toda red de ordenación para n elementos debe incluir al menos $\lceil lg\, n! \rceil$ comparadores. Por ejemplo, toda red de ordenación para 16 elementos tiene que contener un mínimo de 45 comparadores. La red par-impar de ordenación S_{16} contiene 63 comparadores. Se conoce una red diferente que utiliza solamente 60 comparadores. ¡Parece que aún es mejorable! Tampoco en lo tocante a pro-

fundidad la red par-impar de ordenación es óptima: para 16 entradas, la red par-impar de ordenación tiene una profundidad 10, pero se conoce una red diferente que tiene profundidad 9. Se sabe que existe una clase de redes de ordenación cuya profundidad está en $\Theta(\log n)$ y cuyo tamaño está en $\Theta(n \log n)$, pero todavía no ha producido redes que resulten útiles en la práctica.

11.7 ORDENACIÓN EN PARALELO

Las secciones anteriores describen la forma de efectuar ordenaciones en paralelo empleando una red de comparadores. Si se simula cada comparador mediante un procesador, se obtiene un algoritmo que puede ejecutarse mediante un computador *p-ram* CREW. Este algoritmo puede ordenar n elementos en un tiempo que está en $\Theta(\log^2 n)$, empleando $\Theta(n \log^2 n)$ procesadores. (Esto se sigue directamente del resultado del problema 11.17). En esta sección se esboza un algoritmo de ordenación debido a Cole que se puede ejecutar en un *p-ram* CREW, y que ordena n elementos en un tiempo que está en $\Theta(\log n)$ empleando un número de procesadores que está en $\Theta(n)$. El trabajo efectuado está, por tanto, en $\Theta(n \log n)$. Tal como veremos en la Sección 12.2.1, este algoritmo paralelo es óptimo, al menos en lo tocante a la ordenación por comparación.

A lo largo de esta sección supondremos por sencillez que tenemos n elementos distintos, que deben clasificarse en orden ascendente, y donde n es una potencia de 2; en caso contrario, añadimos el número necesario de elementos mudos. En esencia, el algoritmo paralelo de Cole es una ordenación por fusión basada en árboles. Puede servir de ayuda compararlo con la ordenación por fusión secuencial ordinaria que se describe en la Sección 7.4.1. El algoritmo emplea un árbol binario completo con n hojas. Inicialmente, se dispone un elemento a ordenar en cada hoja. Entonces el cálculo progresa ascendiendo por el árbol, nivel por nivel, desde las hojas hasta la raíz. En cada nodo interno, los subconjuntos ordenados producidos por sus hijos se van fusionando. Supongamos que esta fusión se pueda efectuar en un tiempo que esté en $\Theta(M(n))$, y que todas las fusiones del mismo nivel del árbol se puedan efectuar en paralelo. Dado que hay $\lg n$ niveles de nodos internos en el árbol, todo el proceso completo de ordenación se puede efectuar en un tiempo que está en $\Theta(M(n)\log n)$.

Si no se dispone de información adicional acerca de dos sucesiones ordenadas cada una de las cuales contiene m elementos, se puede demostrar que con m procesadores el tiempo requerido para fusionar las secuencias está en $\Omega(\log\log m)$. Empleando esta aproximación, esperaríamos que una ordenación por fusión basada en árboles requiriera un tiempo que estuviese al menos en $\Omega(\log n \log\log n)$. La idea de Cole era proporcionar tan solo la información adicional necesaria para que fuera posible fusionar dos sucesiones ordenadas en un tiempo que estuviera en $O(1)$, de tal forma que el algoritmo global se ejecutase en un tiempo en $O(\log n)$.

La sección siguiente agrupa algunas definiciones necesarias, y después se esboza el algoritmo.

11.7.1 Preliminares

Hay que ordenar n elementos. A medida que progresa el algoritmo, se almacenan diferentes subconjuntos de estos n elementos por orden ascendente en vectores ordenados. En lo que sigue, dado que todos los vectores de interés están ordenados, nos limitamos a llamarlos vectores, y damos por supuesto que están ordenados los elementos que contienen. Utilizaremos letras minúsculas para denotar los elementos, y letras mayúsculas para denotar vectores.

Sean a, b y c tres elementos, con $a < c$. Decimos que b está *entre* a y c si $a \leq b < c$. También diremos que a y c *rodean* a b.

Sea L un vector de elementos. Si a es un elemento de L, entonces el *rango* de a en L se define de forma sencilla: el menor elemento de L tiene rango 1, el próximo en tamaño tiene el rango 2, y así sucesivamente. Suponemos tácitamente que todos los vectores L se aumentan con dos elementos invisibles, a saber, $-\infty$ que tiene rango 0 y $+\infty$, que tiene rango $|L|+1$. Esto nos permite definir el rango de un elemento de un vector con respecto a otro (que a veces se denomina *rango cruzado*) en la forma siguiente. Sean L y J dos vectores es, y sea b un elemento de J. Si a y c son los dos elementos consecutivos de L que rodean a b (y existen necesariamente, por existir los dos elementos invisibles de L), entonces se define el rango de b en L como igual al rango de a en L. Denotaremos mediante J/L el conjunto de rangos respecto a L de todos los elementos de J.

Una vez más, sean J y L dos vectores de elementos, y sean ahora a y b dos elementos consecutivos de L (incluyendo los elementos invisibles). Definimos el intervalo *inducido* por estos elementos en la forma $[a, b]$. Un elemento x pertenece a este intervalo si $a \leq x < b$, esto es, si a y b rodean a x. Diremos que L *cubre* a J si todo intervalo inducido por elementos consecutivos de L contiene como máximo tres elementos de J. (En esta ocasión, los elementos invisibles de J *no* se incluyen.)Por ejemplo, si L contiene los elementos 10, 20 y 30, mientras que J contiene a 1, 7, 22, 23, 26 y 35, entonces L cubre a J: el intervalo inducido por los elementos $-\infty$ y 10 de L contiene dos elementos de J, el intervalo inducido por 10 y 20 no contiene ninguno, y así sucesivamente. Sin embargo, si K contiene los elementos 11, 12, 14, 17, 22 y 41, entonces L no cubre a K porque el intervalo inducido por los elementos de 10 a 20 de L contiene más de tres elementos de K.

Utilizaremos el símbolo & para denotar la operación de fusión de dos vectores ordenados. Supongamos que L cubre a J, y que M cubre a K. En contra de lo que podría esperarse, no es necesariamente cierto que $L\&M$ cubra a $J\&K$; véase el problema 11.19. Finalmente, para todo vector L, $r(L)$ denota el vector que se obtiene tomando todos los cuartos elementos de L que ocupan posiciones múltiplos de cuatro. Si L contiene menos de cuatro elementos $r(L)$ está vacía.

11.7.2 La idea clave

Ahora daremos una descripción informal del método para fusionar dos vectores en un tiempo que está en $O(1)$. Sean J, K y L tres vectores ordenados, y supongamos que L cubre tanto a J como a K. La situación es la ilustrada en la figura 11.17.

Dado que L cubre a J, todo intervalo inducido por dos elementos consecutivos de L, incluyendo los inducidos por los elementos invisibles, contiene al menos tres elementos de J; lo mismo se puede decir para K. Si conocemos los rangos J/L, K/L, L/J y L/K, es fácil determinar qué elementos se encuentran en cada intervalo.

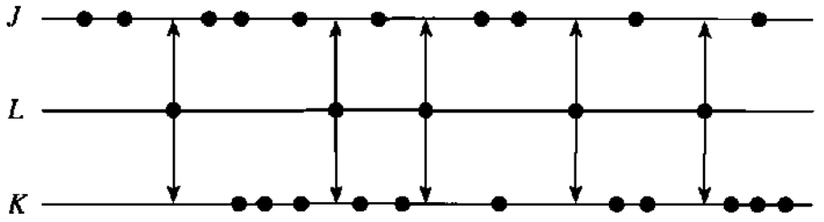


Figura 11.17. Fusión de J y K con ayuda de L

Los elementos de J situados en el primer intervalo inducido por L se pueden fusionar con los de K situados también en el primer intervalo en un tiempo que está en $O(1)$, porque hay como mucho seis de ellos en total. Lo mismo se puede decir de los elementos de J y de K situados en el segundo intervalo inducido por L, y así sucesivamente. Si disponemos de un número suficiente de procesadores, todas estas fusiones de un máximo de seis elementos se pueden efectuar en paralelo, así que será posible terminarlas todas en un tiempo que esté en $O(1)$. Para obtener $J\&K$, lo único que resta por hacer es concatenar los resultados de fusionar cada intervalo por separado. De esta manera, toda la fusión se puede completar en tiempo constante.

Llamaremos a esta operación *fusión con ayuda*. En el ejemplo anterior, decimos que los vectores J y K se fusionan con ayuda de L.

11.7.3 El algoritmo

Los n elementos que hay que ordenar se sitúan inicialmente en las hojas de un árbol binario completo. En un nodo interno típico v de este árbol, la tarea es calcular L_v, un vector que contiene todos los elementos del subárbol cuya raíz es v. en cada fase del cómputo en el nodo v, hay un vector A_v que contiene un subconjunto de los elementos de L_v; A_v se recalcula en cada fase. Más exactamente, sea $A_v(t)$ el vector en cuestión al principio de la fase t, $t = 0, 1,....$, y sea $A_v(t + 1)$ el vector creado durante esa fase. En general, $A_v(t + 1)$ es de un tamaño doble al de $A_v(t)$, hasta que finalmente $A_v(t + 1) = L_v$. Un nodo v en el cual $A_v(t) \neq L_v$ se denominará *activo*, mientras que un nodo v en el cual $A_v(t) = L_v$ se denomina *completo*. los vectores asociados con los hijos de v reciben los sufijos x e y: no importa a qué hijo se refiera cada sufijo. En cada fase están implicadas otros tres vectores: $B_v(t)$ es un vector que se transmite subiendo hasta el padre de v, y $B_x(t)$ y $B_y(t)$ son dos vectores que se reciben de los hijos de v.

En primera aproximación, el cálculo efectuado durante cada fase en todos los nodos internos v comprende las tres fases siguientes (véase la figura 11.18):

1. Calcular el vector $B_v(t) \leftarrow r(A_v(t))$ y enviarla árbol arriba al padre de v.

2. Leer las dos matrices $B_x(t)$ y $B_y(t)$ que los hijos de v acaban de enviar por el árbol.
3. Calcular $A_v(t+1) \leftarrow B_x(t)$ & $B_y(t)$. Esta operación de fusión se efectúa en tiempo constante, según se indicaba más arriba, con ayuda del vector $A_v(t)$, que, tal como veremos, cubre tanto a $B_x(t)$ como a $B_y(t)$.

Hay diferencias en las tres fases según el nodo v sea activo o completo. Sin embargo, para nuestros propósitos en este momento es innecesario dar más detalles; basta con tener en cuenta que

◊ en la fase 0, los nodos del nivel 1 leen los valores enviados por sus hijos, que son hojas, fusionan estos valores y pasan a ser completos,
◊ pasadas tres fases después de que un nodo llega a ser completo, su padre pasa a su vez a ser completo.

Dado que hay lg n niveles de nodos internos en el árbol, concluimos que el algoritmo posee 3 lg n −2 fases.

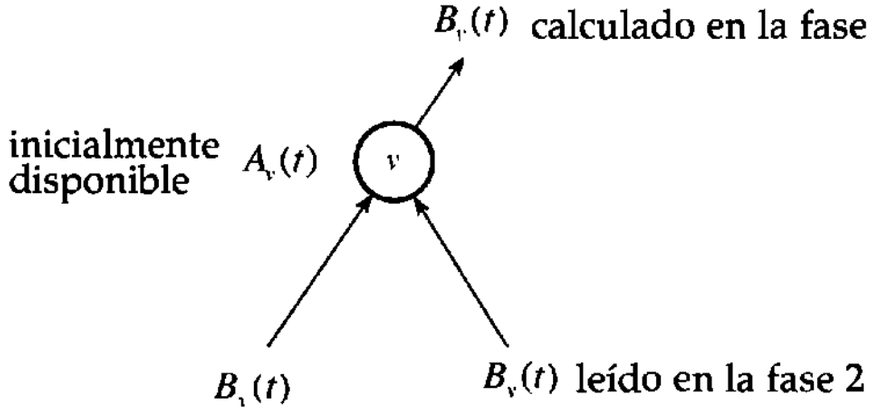


Figura 11.18. Fase t del nodo v

Queda por demostrar que las fusiones requeridas por la fase 3 de cada etapa se pueden efectuar en un tiempo que está en $O(1)$. Combinado con el resultado anterior, esto mostrará que el algoritmo completo se ejecuta en un tiempo que está en $O(\log n)$.

11.7.4 Un esbozo de los detalles

Para demostrar que en cualquier nodo interno v con hijos x e y en cualquier fase t se pueden fusionar los vectores$B_x(t)$ y $B_y(t)$ en un tiempo $O(1)$ con ayuda de $A_v(t)$, es preciso demostrar dos cosas: en primer lugar, que $A_v(t)$ cubre tanto a $B_x(t)$ como a $B_y(t)$; y en segundo lugar que los rangos $A_v(t)/B_x(t)$, $A_v(t)/B_y(t)$, $B_x(t)/A_v(t)$ y $B_y(t)/A_v(t)$ son conocidos, o bien se pueden determinar rápidamente.

Es bastante sencillo demostrar la propiedad de cobertura requerida. Hay un argumento largo pero no difícil que muestra que todos los rangos necesarios se pueden obtener en $O(1)$ siempre y cuando se asigne un procesador a cada elemento del vector $A(t)$ en todos los nodos activos, otro a cada elemento del vector $B(t)$ en todos los nodos activos, y por último otro a cada elemento del vector $A(t+1)$ en todos los nodos activos. Por tanto, todas las operaciones requeridas por una sola fase del algoritmo se pueden efectuar en tiempo en $O(1)$ siempre y cuando esté disponible un número de procesadores suficiente para asignar uno a cada uno de los elementos de los vectores $A(t)$, $B(t)$ y $A(t+1)$ en todos los nodos activos.

Para un nodo v, el tamaño del vector $A_v(t)$ es la cuarta parte del tamaño de los vectores $A_x(t)$ y $A_y(t)$ que tienen sus hijos, siempre y cuando los hijos todavía no sean completos. Por tanto el tamaño total de los vectores $A(t)$ en el nivel de v es de un octavo del tamaño de los vectores $A(t)$ en el nivel de sus hijos, siempre y cuando los hijos no sean completos. (Hay dos veces más hijos que padres). Aún cuando no hemos dado los detalles, esto también es cierto en la fase en que los hijos pasan a ser completos por primera vez. en la segunda fase en que los hijos son completos, el tamaño de $A_v(t)$ en v es la mitad del tamaño de $A_x(t)$ y $A_y(t)$ en sus hijos, así que el tamaño total de estos vectores en el nivel de v es la cuarta parte del tamaño que tienen en el nivel de los hijos; de forma similar, en la tercera fase en que los hijos son completos, el tamaño total de los vectores $A(t)$ en el nivel de v es la mitad del tamaño total de estos vectores en el nivel de los hijos.

Dado que nunca puede haber más de n elementos en todos los vectores $A(t)$ en un mismo nivel, el número total de elementos en todos los vectores $A(t)$ a lo largo de todo el árbol no puede superar el máximo de tres valores: $n + n/8 + n/16 + \ldots = 8n/7$ cuando hay n elementos en el nivel cuyos nodos acaban se alcanzar el estado de completos, $n + n/4 + n/32 + \ldots = 9n/7$ en la etapa siguiente, $n + n/2 + n/16 + \ldots = 11n/7$ al cabo de otra etapa más. Por tanto, el tamaño total de todos los vectores $A(t)$ está acotado por $11n/7$. De manera similar, se puede mostrar que el tamaño total de todos los vectores $B(t)$, y por tanto el tamaño total de todos los vectores $A(t+1)$ en todos los nodos activos no es mayor que $8n/7$. El número total de elementos de todos los vectores $A(t)$, $B(t)$ y $A(t+1)$ en todos los nodos activos no será mayor, por tanto, que $27n/7$.

Concluimos finalmente que el algoritmo se puede ejecutar en un tiempo que está en $O(1)$ empleando un número de procesadores que está en $O(n)$. Tal como se ha descrito, el algoritmo no tiene conflictos de escritura, aunque pueden producirse lecturas simultáneas.

11.8 COMENTARIOS ACERCA DE LAS P-RAM EREW Y CRCW

A lo largo de este capítulo hemos supuesto que nuestros algoritmos se van a ejecutar en un *p-ram* CREW, esto es, en una máquina que permite a varios procesadores leer en una misma posición simultáneamente, pero que no permite la escri-

tura simultánea en una misma posición. Intuitivamente, parece probable que una máquina de estas características debería ser más potente que una *p-ram* EREW, y menos potente que una *p-ram* CRCW. Ahora daremos dos ejemplos sencillos que confirman esta intuición.

Para mostrar que una *p-ram* CREW puede tener menor rendimiento que una *p-ram* EREW, considere el problema siguiente. Un árbol binario contiene n nodos, y todo nodo i salvo la raíz está enlazado con su antecesor mediante un puntero $p[i]$. En la raíz, el puntero p tiene el valor especial **nil**. Todo nodo i posee además un segundo puntero $r[i]$. Deseamos fijar el valor de r en todos los nodos de tal manera que apunte a la raíz del árbol. Considere el siguiente algoritmo paralelo para hacerlo.

```
procedimiento buscar- raíz(T)
    {T es un árbol binario con n nodos}
    para todo par ordenado (i, j) hacer en paralelo
        si p[j] = nil entonces r[i] ← j
```

Aquí tanto i como j varían entre 1 y n, así que necesitamos n^2 procesadores para ejecutar la instrucción **si** en paralelo para todo para de nodos. Suponiendo que dispongamos de ellos, la instrucción se puede ejecutar en paralelo par todos los pares de nodos en un *p-ram* CREW. Cuando se evalúa la condición, n procesadores leen el valor de $p[j]$ para todo j; sin embargo, se admiten las lecturas concurrentes, así que esto no es problema. En lo tocante a la instrucción de asignación, sólo un procesador para cada valor de i llega a la conclusión de que la condición es verdadera, así que para cada valor de i sólo un procesador le asigna un valor a $r[i]$. Por tanto, no hay conflictos de escritura. Las operaciones implicadas son elementales, así que el algoritmo puede ejecutarse en un tiempo que está en $O(1)$ empleando n^2 procesadores CREW.

Por otra parte, un sencillo argumento muestra que ningún algoritmo para un procesador EREW puede producir el resultado apetecido en un tiempo que esté en $O(1)$, independientemente del número de procesadores disponibles. Para cada paso en una máquina de estas características, sólo un elemento de información almacenado en memoria, como máximo, puede ser leído y copiado por un único procesador. Por tanto, el número de posiciones de memoria que puede contener este elemento de información tan sólo puede doblarse, como máximo, en cada paso. En el problema anterior, la identidad de la raíz es conocida inicialmente por un nodo como máximo. Para copiar esta información y almacenarla en los n nodos, se necesita por tanto un mínimo de $\lceil \lg n \rceil$ pasos. Por tanto, cualquier algoritmo para este problema que utilice procesadores EREW requiere un tiempo que está en $\Omega(\log n)$.

Un segundo problema sencillo servirá para mostrar que una máquina CRCW puede tener mayor rendimiento que una máquina CREW. El problema consiste en hallar el máximo de n números que están almacenados en un vector $A[1..n]$. Considere el algoritmo siguiente.

```
función hallar max(A[1..n])
    matriz M[1..n] de Boolean
    para 1 ≤ i ≤ n hacer en paralelo M[i] ← verdadero
    para todos los pares ordenados (i,j) hacer en paralelo
        si A[i] < A[j] entonces M[i] ← falso
    para 1 ≤ i ≤ n hacer en paralelo
        si M[i] entonces max ← A[i]
    devolver max
```

Tal como antes, i y j varían entre 1 y n, así que necesitamos n^2 procesadores para ejecutar la primera instrucción **si** en paralelo para todas las parejas de elementos del vector A. Suponiendo que estén disponibles, la instrucción se puede ejecutar en un tiempo que está en $O(1)$. Sin embargo, aquí pueden intentar escribir simultáneamente en $M[i]$ hasta $n-1$ procesadores. La máquina, por tanto, debe de admitir las operaciones concurrentes de escritura. Obsérvese que si están implicados varios procesadores, todos ellos intentarán escribir el mismo valor: a saber, *falso*. Esta restricción adicional suele imponérsele al modelo CRCW. Tras la ejecución de la primera instrucción **si**, $M[i]$ es *verdadero* si y sólo si $A[i]$ es igual al mayor valor de A. (Puede haber varios de estos elementos.) Finalmente, la segunda instrucción **si** escribe este valor máximo en *max*. Una vez más, es posible que varios procesadores intenten escribir simultáneamente en *max*, pero si lo hacen, entonces todos ellos intentarán escribir el mismo valor. También esta instrucción se puede ejecutar en un tiempo que está en $O(1)$. Por tanto, todo el algoritmo se puede ejecutar en un tiempo que está en $O(1)$ en un *p-ram* CRCW que utilice n^2 procesadores.

Aun cuando el argumento no resulta sencillo en este caso, se puede demostrar que cualquier algoritmo (como los que se describían anteriormente en este capítulo) para hallar el máximo de n elementos utilizando un *p-ram* CREW debe emplear un tiempo que esté en $\Omega(\log n)$, independientemente del número de procesadores que estén disponibles. Por tanto, un *p-ram* CRCW es más potente que un *p*-ram CREW.

11.9 COMPUTACIÓN DISTRIBUIDA

En la Sección 11.1 definíamos un modelo de computación paralela llamado «modelo de instrucción única y múltiples flujo de datos», en el cual los procesadores están más o menos sincronizados. Supongamos que relajamos esta instrucción, y permitimos que cada procesador opere sobre sus propios datos, a su propia velocidad, y empleando su propio programa. Cuando tenga que informar acerca de algo interesante, enviará un mensaje a sus colegas, y quizá reciba mensajes de ellos en algunas ocasiones. Éste es el modelo de *múltiples instrucciones y múltiples flujo de datos*. Los dos modelos suelen denominarse SIMD y MIMD, respectivamente.

Con el modelo SIMD, aunque no sea una necesidad lógica, suele resultar conveniente mantener muy próximos a los procesadores implicados en la ejecución de un algoritmo paralelo, quizá incluso en un mismo dispositivo físico. En caso con-

trario, el coste adicional asociado a su sincronización puede resultar prohibitivo. Una vez que se relajan las restricciones, sin embargo, ya no hay razón para que todos los procesadores implicados en la ejecución de un algoritmo paralelo tengan que hallarse todos ellos en una misma ubicación. Si los mensajes se intercambian con frecuencia relativamente escasa, los procesadores pueden estar en cualquier parte del mundo, y los mensajes se pueden enviar por correo electrónico. La ejecución de un algoritmo de este tipo es un ejemplo de lo que se llama *computación distribuida*. Dos ejemplos sorprendentes de esta técnica tienen que ver respectivamente con la factorización de enteros muy grandes y al problema del viajante.

Para el primer ejemplo, considérese el algoritmo de Las Vegas para factorizar enteros muy grandes, que se describe en la Sección 10.7.4. El algoritmo depende esencialmente de hallar un número suficiente de enteros con una propiedad especial (sus cuadrados módulo n tienen que ser k-suaves (véase la descripción del algoritmo en la Sección 10.7.4), donde los candidatos se seleccionan aleatoriamente. Si están disponibles varios procesadores, es evidentemente posible hacer que uno de ellos recoja los enteros necesarios, mientras que los otros van cribando los posibles candidatos. Cada procesador puede trabajar independientemente de los demás, salvo quizá algunos acuerdos para evitar una innecesaria duplicación de esfuerzos. Cuando se encuentra un entero apropiado, el procesador en cuestión puede enviar un mensaje con la nueva información al encargado de la colección, hasta que finalmente se haya encontrado un número suficiente de enteros. Si el hallazgo de un entero adecuado es un suceso relativamente infrecuente, entonces el correo electrónico tiene una velocidad perfectamente suficiente para transmitir los mensajes.

Empleando este tipo de técnicas, Lenstra y Manasse diseñaron un experimento que consistió en el reclutamiento de voluntarios con acceso a Internet, en suministrarles los programas necesarios y en recoger los resultados a medida que estos iban apareciendo. El experimento comenzó en el verano de 1987, empleando el método de curva elíptica para la factorización, que no describimos en este libro. En 1988 cambiaron a una variante del algoritmo de criba cuadrática que se esboza en la Sección 10.7.4. Sus programas estaban diseñados para ejecutarse en una estación de trabajo durante aquellos períodos en que normalmente estaría desocupada —por la noche, o durante los fines de semana, por ejemplo— de tal manera que la potencia de cálculo empleada era prácticamente gratuita. Estiman que tuvieron acceso al equivalente de unos 1.000 millones de instrucciones por segundo de potencia de cálculo sostenida, lo cual les permitía factorizar enteros de 100 cifras en alrededor de una semana. Esperaban que el tiempo de cálculo se duplicase aproximadamente por cada tres cifras que se añadieran al número que había que factorizar. Véase la Sección 10.7.4 para un ejemplo reciente de un éxito todavía más impresionante para un número de 129 dígitos.

En el caso del problema del viajante, todos los programas de computadora eficientes se basan en un método que consiste en hallar planos de corte adecuados. (No se preocupe si no sabe lo que son.) Una vez más, se pueden utilizar varios procesadores para buscar planos de corte independientemente. En 1993, un equipo de

cuatro personas resolvía un problema de 4.461 ciudades después de calcular durante 28 noches en paralelo, en una red de 75 máquinas. Estiman que el cálculo en cuestión habría requerido casi dos años si se hubiese ejecutado en una sola estación de trabajo.

11.10 PROBLEMAS

Problema 11.1. Demostrar que el algoritmo *aplanar* es correcto.

Problema 11.2. Demostrar que el mínimo de n elementos almacenados en un vector se puede hallar en un tiempo que está en $\Theta(\log n)$ empleando $\Theta(n / \log n)$ procesadores.

Problema 11.3. Con las definiciones de la Sección 11.2.2, demostrar que cuando el algoritmo *operpar* finaliza, el valor de d para el primer elemento de L es $X_{1,n}$, el valor de d para el segundo elemento de L es $X_{2,n}$, y así sucesivamente, hasta el valor de d para el último elemento que será $X_{n,n}$.

Problema 11.4. Escriba un algoritmo similar a *operpar* de la Sección 11.2.2, salvo que cuando termine el valor de d para el primer elemento de L debe ser $X_{1,1}$, el valor de d para el segundo elemento de L debe ser $X_{1,2}$, y así sucesivamente, hasta el valor de d para el último elemento de L que será $X_{1,n}$.

Problema 11.5. Escribir un algoritmo similar a *operpar* de la Sección 11.2.2, salvo que admite dos parámetros: una lista y un valor. Suponga que el operador $\circ$ admite tres parámetros, y no dos. Dos de ellos son punteros a elementos de la lista, y el tercero es un valor k. Cuando los dos elementos de la lista tienen valores menores o iguales que k, $\circ$ proporciona un puntero al elemento que tenga mayor valor; sólo cuando uno de los elementos tenga un valor menor o igual que k, $\circ$ proporcionará un puntero a este elemento; cuando ninguno de ellos tenga un valor menor o igual que k, $\circ$ proporciona el valor **nil**. El algoritmo debe proporcionar un puntero a un elemento de la lista cuyo valor sea k si existe; en caso contrario, debe de proporcionar un puntero al elemento que tenga el mayor valor que no supere a k. El algoritmo es una especie de "búsqueda binaria" en una lista enlazada, que no se puede hacer secuencialmente.

Problema 11.6. Demostrar que si se tiene un algoritmo paralelo eficiente (que emplee un número polinómico de procesadores y requiera un tiempo polilogarítmico) para algún problema, entonces se puede hallar un algoritmo secuencial eficiente (que requiera un tiempo polinómico) para el mismo problema.

Problema 11.7. Demostrar que si $p > q$ entonces $p / q \leq \lceil p / q \rceil < 2p / q$.

Problema 11.8. Demostrar que el algoritmo *caminospar* se puede ejecutar empleando $\Theta(n^3 / \log n)$ procesadores que requieren un tiempo que está en $\Theta(\log^2 n)$.

Problema 11.9. Dibujar figuras que ilustren el progreso del algoritmo *compcons* aplicado al grafo de la figura 11.19.

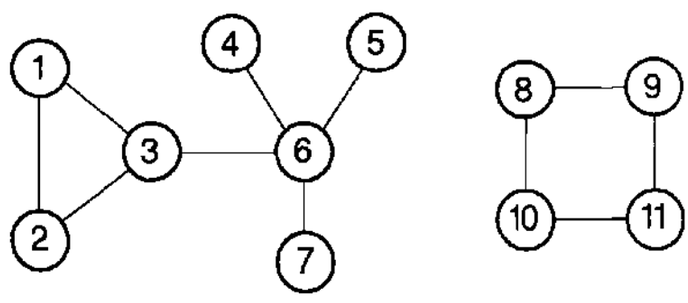


Figura 11.19. Un grafo

Problema 11.10. Dar un ejemplo de una expresión con cinco operandos que requiera cuatro iteraciones del algoritmo sencillo de la Sección 11.5 para evaluarla.

Problema 11.11. Si $f(x) = (a_1x + b_1)/(c_1x + d_1)$, $g(x) = (a_2x + b_2)/(c_2x + d_2)$ y $g(f(x)) = (a_3x + b_3)/(c_3x + d_3)$, hallar las expresiones de a_3, b_3, c_3 y d_3 en términos de a_1, a_2, b_1, b_2 etc.

Problema 11.12. Demostrar que dos operaciones de corte (véase la Sección 11.5) que involucren a las hojas C y C' no interfieren entre sí si C y C' son hojas no consecutivas del árbol de expresiones, y si ambas son o bien hijos izquierdos o bien hijos derechos.

Problema 11.13. Dibujar la expresión

$$7 \times (4 + (64/(10-)3 \times (13-(6+5))))))$$

en forma de un árbol de expresión con hojas numeradas de izquierda a derecha, *e* ilustrar el funcionamiento del algoritmo de evaluación paralela de la Sección 11.5 aplicado a este árbol.

Problema 11.14. Enunciar y demostrar un principio cero-uno para las redes de fusión.

Problema 11.15. Demostrar que el tamaño y profundidad de las redes de fusión F_n descritas en la Sección 11.6.2 son $1 + n \lg n$ y $1 + \lg n$ respectivamente.

Problema 11.16. Mostrábamos en la Sección 11.6.2 la forma de construir redes de fusión F_n cuando n es una potencia de 2. Extender este resultado, mostrando la forma de construir una red de fusión F_n para cualquier $n > 0$. En términos de n, ¿cuáles son el tamaño y la profundidad de las redes resultantes?

Problema 11.17. Demostrar que el tamaño y la profundidad de las redes S_n descritas en la Sección 11.6.3 son $n - 1 + n(\lg^2 n - \lg n)/4$ y $(\lg^2 n + \lg n)/2$, respectivamente

Problema 11.18. Considere las redes definidas en la forma siguiente para $n \geq 2$. La red contiene $n(n-1)/2$ comparadores organizados como ladrillos en una pared, tal como se ilustra en la Figura 11.20: hay $\lceil n/2 \rceil$ comparadores entre las entradas 1 y 2, $\lceil n/2 \rceil$ comparadores entre o detrás de las entradas 2 y 3, otros $\lceil n/2 \rceil$ inmediatamente debajo del primer grupo entre las entradas 3 y 4, y así sucesivamente. Demostrar que estas redes son redes de ordenación válidas. En términos de n, ¿cuál es su profundidad?

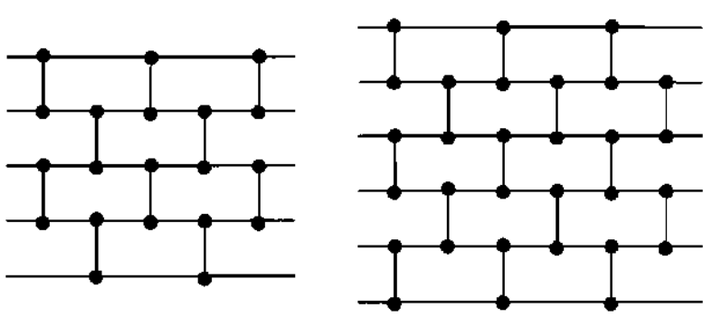


Figura 11.20. Dos redes

Problema 11.19. Sean J, K, L y M cuatro vectores ordenados tales que L cubre a J y M cubre a K, donde la cobertura se ha definido en la Sección 11.7.1. Dar un ejemplo que muestre que $L\&M$ no cubre necesariamente a $J\&K$.

Problema 11.20. Señalar los lugares de la ordenación por fusión paralela de Cole en los que pueden producirse operaciones de lectura simultáneas.

Problema 11.21. Suponga que desea escribir un algoritmo como *distpar* de la Sección 11.2.2, empleando duplicación de punteros,

pero en una lista de longitud desconocida. En lugar de repetir el paso de duplicación en log n ocasiones, podríamos intentar utilizar alguna estructura como la siguiente

repetir
paso de duplicación
hasta que $s[i] = s[s[i]]$ para todo i de L

Esto plantea un problema para el control global de nuestra máquina paralela: ¿cómo pueden saber todos los procesadores que ha terminado el bucle? Demostrar que se puede hacer (a) en un *p-ram* EREW, (b) en un *p-ram* CREW y (c) en un *p-ram* CRCW. ¿Cuánto tiempo se necesita para comprobar la terminación del bucle en cada caso?

11.11 REFERENCIAS Y TEXTOS MÁS AVANZADOS

Entre los libros introductorios acerca del procesamiento paralelo se cuentan Gibbons y Rytter (1988) y Akl (1989). Lakshmivarahan y Dhall (1990) contiene un capítulo útil acerca de los multiprocesadores y multicomputadoras. Leighton (1992) abarca las relaciones entre la arquitectura de las máquinas disponibles comercialmente y los algoritmos paralelos eficientes para toda una gama de problemas. Reif (1993) es más avanzado.

El teorema de Brent procede de Brent (1974). El algoritmo paralelo para buscar caminos mínimos que se describe en la Sección 11.4.1 sigue esencialmente a Kucera (1982). El algoritmo paralelo para hallar los componentes conexos de un grafo descrito en la Sección 11.4.2 procede de Hirschberg, Chandra y Sarwate (1979). El algoritmo óptimo que se menciona pero no se describe al final de esta sección se hallará en Chin, Lam y Chen (1982). Se presentan más algoritmos paralelos para problemas de grafos en Quinn y Deo (1984).

El ejemplo de evaluación en paralelo de expresiones matemáticas sencillas dado en la Sección 11.5 sigue a Gibbons y Rytter (1988). Hay más información acerca de este problema en Brent (1974).

En Batcher (1968) se describen varias clases de redes de ordenación, y Knuth (1973) contiene una sección acerca de este asunto, incluyendo una solución del Problema 11.18. El algoritmo paralelo de ordenación esbozado en la Sección 11.7 es de Cole (1988). Se hallará más acerca de la ordenación en paralelo en Bitton, DeWitt, Hsiao y Menon (1984), mientras que Akl (1985) da veinte algoritmos diferentes para arquitecturas concretas de máquinas paralelas.

El ejemplo de factorización de enteros muy grandes a través de correo electrónico que se presenta en la Sección 11.9 procede de Lenstra y Manasse (1990). La resolución del problema del viajante para una red de 4.461 ciudades fue descrita por Vasek Chvátal en una conferencia pronunciada en Octubre de 1993. Para comprender mejor las clases de arquitecturas de máquinas paralelas que se emplean en la actualidad, véase Duncan (1990), un texto programado que revisa aproximaciones alternativas del procesamiento en paralelo. Un diseño que ha alcanzado especial éxito se describe en Tucker y Robertson (1988).

Capítulo 12

Complejidad Computacional

Hasta el momento, nos ha interesado la *Algoritmia*, que se ocupa del diseño y análisis sistemático de algoritmos específicos, cada uno de los cuales será más eficiente que sus predecesores, para resolver algún problema dado. Sin embargo, un libro sobre algoritmia estaría incompleto sin una introducción a su disciplina gemela: la *complejidad computacional*. Este campo, que tiene una trayectoria paralela a la de la Algoritmia, considera globalmente todos los posibles algoritmos para resolver un problema dado. Esto incluye aquellos algoritmos en los que ni siquiera se ha pensado todavía.

Empleando la algoritmia, podemos demostrar expresando y analizando un algoritmo explícito que el problema objeto de nuestro estudio se puede resolver en un tiempo que está en $O(f(n))$ para alguna función $f(n)$ que intentamos reducir lo más posible. Al emplear la complejidad, por otra parte, intentamos hallar una función $g(n)$ tan grande como sea posible para la cual podamos demostrar que *cualquier* algoritmo capaz de resolver correctamente nuestro problema en todos sus casos debe necesariamente requerir un tiempo que esté en $\Omega(g(n))$. A esta función $g(n)$ se le da el nombre de *cota inferior* de la complejidad del problema. Quedamos completamente satisfechos cuando $f(n) \in \Theta(g(n))$ dado que entonces sabemos que hemos encontrado el algoritmo más eficiente posible, salvo quizá en lo tocante a cambios en la constante multiplicativa oculta. Desafortunadamente, esta felicidad no nos llega con frecuencia en nuestro estado actual de ignorancia. Sin embargo, de vez en cuando podemos encontrar, incluso, el número exacto de veces que es precisa una operación —como la comparación— para resolver el problema. En este capítulo presentamos tan sólo unas pocas de las principales técnicas y conceptos que se emplean en el estudio de la complejidad computacional: argumentos de la teoría de la información, argumentos del adversario, reducción y NP- compleción. También veremos que las técnicas de complejidad pueden ser útiles para el diseño de algoritmos.

12.1 INTRODUCCIÓN: UN EJEMPLO SENCILLO

Considere como ejemplo introductorio el juego de las veinte preguntas. Su amigo selecciona un entero positivo menor que un millón, que usted tiene que adivinar. Se le permite un máximo de 20 preguntas del tipo «sí» o «no». Su amigo tiene que ser capaz de decidir sin ambigüedad si responde «sí» o «no» a cada una de sus preguntas. Por ejemplo, se le puede preguntar: «¿Es el número primo?». Si está familiarizado con este juego, utilizará un enfoque de divide y vencerás para resolver este acertijo, dividiendo por dos el número de candidatos a número misterioso con cada una de sus preguntas. Por tanto, la primera pregunta va a ser «¿Está el número entre 1 y 500.000?» Es elemental demostrar que siempre se hallará la respuesta con menos de las 20 preguntas permitidas, porque un millón es menos que 2^{20}.

Encontrar este algoritmo, que muestra que 20 preguntas son *suficientes* para resolver el problema, era un tema que correspondía a la algorítmia. El que las 20 preguntas fueran o no *necesarias*, sin embargo, concierne a la complejidad. Es fácil demostrar que nuestro algoritmo utilizará casi todas las 20 preguntas en la mayoría de los números buscados. Sin embargo, no se puede concluir, basándose en la incapacidad de este algoritmo para resolver siempre el acertijo con menos de 20 preguntas, que no sea posible hacerlo mediante un algoritmo más inteligente. Las técnicas que vamos a estudiar a continuación, en particular los argumentos de teoría de la información y los argumentos del adversario, nos permiten demostrar muy fácilmente que ciertamente son necesarias las 20 preguntas. Sin embargo, este problema es tan sencillo que podemos resolverlo con una aproximación elemental.

Sea S_i el conjunto de candidatos al número buscado después de haber hecho la i-ésima pregunta, y sea $k_i = |S_i|$ el número de candidatos restantes. Inicialmente, S_0 contiene todos los enteros positivos hasta un millón, y por tanto $k_0 = 10^6$. Sea Q_i la i-ésima pregunta, que puede depender de las respuestas a las preguntas anteriores, y supongamos que $Q_i(n)$ denota la respuesta a esa pregunta si el número buscado es n. Sea $Y_i = \{n \in S_{i-1} \mid Q_i(n) = \text{"}si\text{"}\}$, y sea $N_i = \{n \in S_{i-1} \mid Q_i(n) = \text{"}no\text{"}\}$. Está claro que $N_i \cap Y_i = \varnothing$, y que $N_i \cup Y_i = S_{i-1}$. Por tanto, $|S_{i-1}| = |Y_i| + |N_i|$ lo cual implica que al menos uno de Y_i o N_i contiene $\lceil k_{i-1}/2 \rceil$ números o más. Dado que el número buscado puede ser cualquiera de los candidatos, no podemos descartar la posibilidad de que la respuesta de su amigo a Q_i le haga tomar el mayor de Y_i o N_i como conjunto de candidatos restantes S_i. Esto es cierto independientemente de lo inteligente que sea usted en su elección de preguntas. Por tanto, es posible que $k_i \geq \lceil k_{i-1}/2 \rceil$ para todo i. Dado que es $k_0 = 10^6$, podría ser que $k_1 \geq 500000$, y por tanto que $k_2 \geq 250000$, y así sucesivamente. Prosiguiendo de esta manera, encontramos que puede ser que $k_{19} \geq 2$. Concluimos que pueden quedar por lo menos dos candidatos al número buscado al cabo de 19 preguntas, y por tanto en el caso peor se necesitan 20 preguntas para resolver el acertijo.

12.2 ARGUMENTOS DE TEORÍA DE LA INFORMACIÓN

Esta técnica se aplica a toda una gama de problemas, especialmente a aquéllos que implican la comparación de elementos. Un *árbol de decisión* es una forma de

representar el funcionamiento de un algoritmo para todos los datos posibles de un tamaño dado. Formalmente, se trata de un árbol binario con raíz. Cada nodo interno del árbol contiene una prueba de algún tipo que se aplica a los datos. Cada hoja contiene una salida, que llamaremos *veredicto*. Un *viaje* por el árbol consiste en partir de la raíz y hacer la pregunta que se encuentra allí. Si la respuesta es «si», el viaje prosigue recursivamente por el subárbol izquierdo; en caso contrario, prosigue recursivamente por el árbol derecho. Finaliza el viaje cuando se alcanza una hoja; el veredicto que se encuentre allí es el resultado del viaje.

Considere una vez más el juego de las 20 preguntas, pero suponga por sencillez que se sabe que el número misterioso está entre 1 y 6. Evidentemente, no se necesitan las 20 preguntas. ¿Cuántas se necesitan realmente? La figura 12.1 muestra un árbol de decisión para este juego. Si el número misterioso buscado es $n = 5$, por ejemplo, la primera pregunta es «¿Es $n \leq 3$?» y seguimos por el subárbol derecho porque la respuesta es «no» (luego sabemos que $n \in \{4, 5, 6\}$). Allí encontramos la pregunta «¿Es $n \leq 5$?» y seguimos por la izquierda porque la respuesta es «si» (luego se sabe que $n \in \{4, 5\}$). Finalmente, se hace la pregunta «¿Es $n \leq 4$?» y se alcanza el veredicto correcto en el lado derecho «$n = 5$» a partir de la respuesta «no».

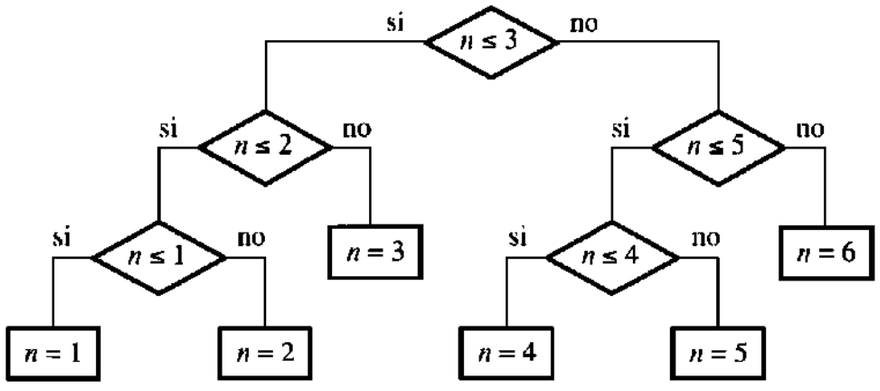


Figura 12.1 Un árbol de decisión para el juego de las tres preguntas

Esto es relevante porque a todo algoritmo determinista para jugar le corresponde un árbol de decisión, siempre y cuando exista un límite para el número de preguntas que puede hacer el algoritmo. A la inversa, se puede pensar en un árbol de decisión como un algoritmo. Supongamos por sencillez que los árboles de decisión están *podados* en el sentido de que se puede acceder a cualquiera de las hojas desde la raíz mediante alguna sucesión consistente de soluciones. Aunque sea un desperdicio, se permite efectuar una pregunta cuya respuesta esté determinada por la sucesión de preguntas y respuestas que hayan llevado a esa pregunta. Por ejemplo, uno puede preguntarse si $A \leq C$ en un nodo que se alcance después de saber que $A \leq B$ y que $B \leq C$: veremos en la figura 12.5 que este comportamiento puede aparecer en algoritmos naturales. Recuerde que la altura del árbol es la distancia entre la raíz y la hoja más distante. Dado que se hace una pregunta para cada nodo interno de esa ruta, la altura del árbol corresponde al número de preguntas que se hacen en el caso peor. Además, el ár-

bol tiene que tener al menos una hoja para cada posible veredicto. (El problema 12.5 ilustra el hecho de que en general puede haber más hojas que veredictos distintos).

Volvamos ahora a la cuestión de jugar el juego original de las veinte preguntas, pero ahora sólo con 19 preguntas. Cualquier solución daría lugar a un árbol de decisión cuya altura fuera como máximo 19, y que tendría que tener al menos un millón de hojas. Esto es imposible porque el árbol de decisión es binario, todo árbol binario de altura k tiene como máximo 2^k hojas (véase el problema 12.1) y 2^{19} es menor que un millón. Concluimos inmediatamente que el juego de las veinte preguntas no se puede resolver con 19 preguntas en el caso peor.

Los árboles de decisión también se pueden utilizar para analizar la complejidad de un problema en el caso promedio, en lugar de en el caso peor. Sea T un árbol binario. Se define la *altura media* de T como la suma de las profundidades de todas las hojas, dividido por el número de hojas. Por ejemplo, el árbol de decisión de la figura 12.1 tiene una altura media de (3 + 3 + 2 + 3 + 3 +2) / 6 = 8 / 3. Del mismo modo que la altura de un árbol de decisión podado nos da el rendimiento del algoritmo correspondiente en el caso peor, su altura media nos da el rendimiento en el caso promedio, siempre y cuando todos los veredictos sean igualmente probables y cada veredicto aparezca exactamente una sola vez como hoja del árbol. Prosiguiendo con nuestro ejemplo simplificado, si todos los enteros entre 1 y 6 tienen igual probabilidad de ser el número buscado, el algoritmo correspondiente a la figura 12.1 hace 8/3 preguntas en el caso promedio. ¿Se puede hacer mejor? ¿Es posible mejorar el rendimiento en el caso promedio, quizá si uno está dispuesto a hacer más preguntas que las necesarias en unos pocos casos?

El teorema siguiente nos dice que si el número buscado se selecciona aleatoriamente entre 1 y 6 según la distribución uniforme, entonces cualquier algoritmo para jugar a este juego tiene que hacer al menos $\lg 6 \approx 2.585$ preguntas en el caso promedio, independientemente del número de preguntas que pueda hacer en algunos casos. Pero claramente el número medio de preguntas hechas por cualquier algoritmo determinista tiene que ser un entero dividido por seis, puesto que el número de preguntas es un entero para cada uno de los seis veredictos equiprobables. La solución dada en la figura 12.1 hace 16/6 preguntas en el caso promedio. Toda mejora haría como mucho 15/6 = 2,5 preguntas. Esto se descarta, puesto que $\lg 6 > 15/6$. Concluimos que nuestro árbol de decisión proporciona un algoritmo óptimo cuando el número buscado está entre 1 y 6, tanto en el caso peor como en el caso promedio. De manera similar, se necesitan 20 preguntas en el caso peor para el juego original con un millón de veredictos, y ningún algoritmo puede hacer menos de $\lg 10^6 \approx 19{,}93$ preguntas en el caso promedio si todos los veredictos son igualmente probables.

Teorema 12.2.1. Todo árbol binario con k hojas tiene una altura media que es como mínimo $\lg k$.

Demostración. Sea T un árbol binario con k hojas. Se define $H(T)$ como la suma de las profundidades de las hojas. Por ejemplo, $H(T) = 16$ para el árbol de la figura 12.1. Por definición, la altura media de T es $H(T)/k$, y por tanto nuestro objetivo es demostrar que es $H(T) \geq k \lg k$. La raíz de T puede tener 0, 1 o 2 hijos; véase la figura 12.2. En el primer caso, la raíz es la única hoja del árbol: $k = 1$ y $H(T) = 0$. En el segundo caso, el único hijo es la raíz de un subárbol A, que también tiene k hojas. La distancia de cualquier hoja a la raíz de T es uno más que la distancia desde esa misma hoja a la raíz de A, así que $H(T) = H(A) + k$. En el tercer caso, T consta de una raíz y dos subárboles A y B con i y k-i hojas, respectivamente, para algún $1 \leq i < k$. Por un argumento similar, obtenemos esta vez que $H(T) = H(A) + H(B) + k$.

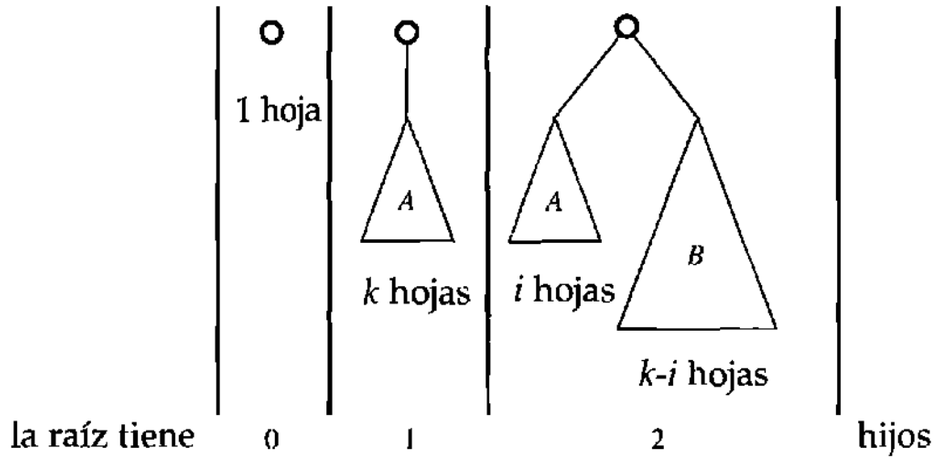


Figura 12.2. Minimización de la altura media de un árbol binario con k hojas

Para $k \geq 1$, se define $h(k)$ como el menor valor posible para $H(X)$ cuando X es un árbol binario con k hojas. En particular, $H(T) \geq h(k)$. Claramente, $h(1) = 0$. Si definimos $h(0) = 0$, la discusión anterior y el principio de optimalidad que se emplea en programación dinámica dan lugar a

$$h(k) = \min_{0 \leq i \leq k} \big(h(i) + h(k-i) + k\big)$$

para todo $k > 1$. A primera vista, esta recurrencia no está bien fundada porque define a $h(k)$ en términos de sí mismo cuando hacemos $i = 0$ o bien $i = k$ en el mínimo. Sin embargo, es imposible que $h(k) = h(k) + k$, así que estos términos no pueden proporcionar el mínimo. Entonces podemos reformular la recurrencia que define $h(k)$

$$h(k) = \begin{cases} 0 & \text{si } k \leq 1 \\ k + \min\limits_{1 \leq i \leq k-1} \big(h(i) + h(k-i)\big) & \text{en caso contrario} \end{cases}$$

Ahora demostraremos por inducción matemática que $h(k) \geq k \lg k$ para todo $k \geq 1$. La base $k = 1$ es inmediata. Sea $k > 1$. Tomemos como hipótesis de inducción que $h(i) \geq i \lg i$ para todo entero positivo i menor que k. Por definición,

$$h(k) = k + \min_{1 \leq i \leq k-1} \big(h(i) + h(k-i)\big)$$

Por la hipótesis de inducción,

$$h(k) \geq k + \min_{1 \leq i \leq k-1} \big(i \lg i + (k-i)\lg(k-i)\big).$$

Para tener a nuestra disposición las herramientas del análisis real para la minimización de funciones, sea g: $[1,k-1] \to \mathbb{R}$ define como $g(x) = x \lg x + (k-x)\lg(k-x)$. Calculando la derivada se obtiene $g'(x) = \lg - \lg (k-x)$, que es cero si y solo si $x = k-x$. Dado que la segunda derivada es positiva, es positiva, $g(x)$ alcanza su mínimo en $x = k/2$. Este mínimo es $g(k/2) = k \lg k - k$. Pero el valor mínimo alcanzado por $g(i)$ cuando i es un entero entre 1 y $k-1$ no puede ser menor que el valor mínimo de $g(x)$ cuando se permite que x sea un número real dentro del mismo intervalo. Por tanto,

$$\min_{i \in 1..k-1} g(i) \geq \min_{x \in [1..k-1]} g(x) \geq k \lg k - k$$

Reuniéndolo todo, alcanzamos la conclusión deseada:

$$H(T) \geq h(k) \geq k + \min_{i \in 1..k-1} g(i) \geq k \lg k$$

y por tanto la altura media de T, que es $H(T)/k$, es al menos $\lg k$.

12.2.1 La complejidad de la ordenación

¿Cuál es el número mínimo de comparaciones necesarias para ordenar n elementos? Por sencillez, contaremos solamente las comparaciones entre elementos que deban de ser ordenados, ignorando aquellas que se puedan hacer para controlar los bucles de nuestro programa. Vimos en la Sección 2.7.2 que la respuesta es ¡"ninguna"! De hecho, la ordenación por urnas no necesita comparación alguna, siempre y cuando esté disponible una cantidad de espacio suficiente. Sin embargo, solo es útil en circunstancias especiales. Para hacer interesante la pregunta, limitaremos nuestra atención a algoritmos de ordenación *basados en comparaciones*: la única operación admisible sobre los elementos que hay que ordenar consiste en compararlos por parejas para determinar si son iguales y, en caso contrario, cual de ellos es el mayor. En particular, no se permite efectuar operaciones aritméticas con los elementos que haya que ordenar. Obsérvese que la ordenación por urnas efectúa una aritmética implícita, porque la indexación dentro de un vector implica añadir el desplazamiento a la dirección base del vector. La diferencia entre permitir operaciones aritméticas sobre los elementos y restringir el algoritmo a compararlos es similar a la diferencia entre la codificación por dispersión y la búsqueda binaria.

Nuestra pregunta es, entones: ¿cuál es el número mínimo de comparaciones necesarias en cualquier algoritmo para ordenar n elementos por comparación? Aunque los teoremas de esta sección son válidos aun cuando consideremos algoritmos probabilistas de ordenación, por sencillez limitaremos nuestra discusión a los algoritmos deterministas. Recurrimos una vez más a los árboles de decisión. Un árbol de decisión para ordenar n elementos es *válido* si asocia a cada posible relación de orden entre los elementos un veredicto compatible con esta relación. Por ejemplo, la figura 12.3 es un árbol de decisión válido para ordenar A, B y C.

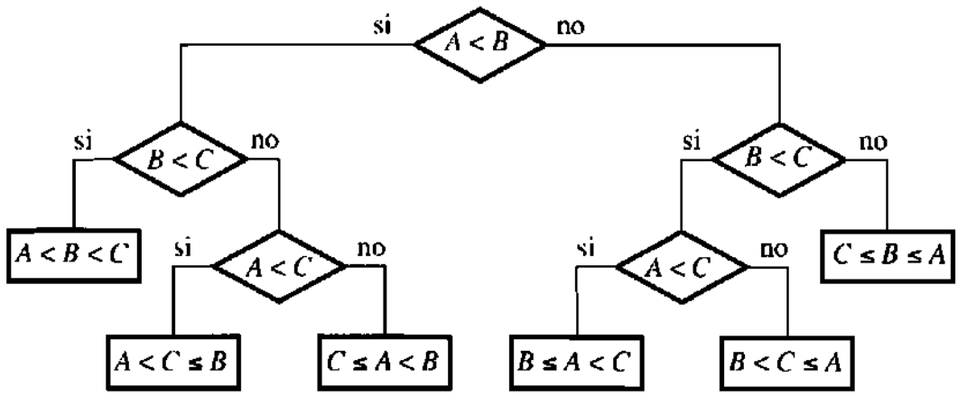


Figura 12.3 Un árbol de decisión válido para ordenar tres elementos

Todo árbol de decisión válido para ordenar n elementos da lugar a un algoritmo de ordenación *in situ (ad hoc)* para el mismo número de elementos. Por ejemplo, al árbol de decisión de la figura 12.3 le corresponde el algoritmo siguiente.

```
procedimiento ordenacioninsitu3(T[1..3])
    A ← T[1]; B ← T[2]; C ← T[3]
    si A < B entonces si B < C entonces {ya están ordenados }
                                    sino si A < C entonces T ← A, C, B
                                                       sino T ← C, A, B
                 sino si B < C entonces si A < C entonces T ← B, A, C
                                                    sino T ← B, C, A
                                              sino T ← C, B, A
```

De manera similar, a todo algoritmo determinista de ordenación por comparación le corresponde, para cada valor de n, un árbol de decisión que es válido para ordenar n elementos. Las figuras 12.4 y 12.5 aumentan los árboles correspondientes a algoritmo de ordenación por inserción (Sección 2.4) y a *ordenación por montículo* (Sección 5.7) cuando hay que ordenar tres elementos. Las anotaciones de los árboles sirven para ayudarnos a seguir el progreso de los algoritmos correspondientes. Observe que *ordenación por montículo* hace comparaciones innecesarias en algunas ocasiones. Por ejemplo, si $B \leq A < C$, el árbol de decisión de la figura 12.5 comprueba primero si $B > A$ (respuesta: no) y después si $C > A$ (respuesta: si). Ahora sería posible establecer el veredicto correcto. Sin embargo, *ordenación por montículo* vuelve a preguntar si $B > A$ antes de alcanzar su conclusión. Para evitar nodos inalcanzables dentro del árbol de decisión, no incluimos la hoja que correspondería a la respuesta contradictoria "si" a esta pregunta inútil; por tanto, el árbol está podado. De esta forma, *ordenación por montículo* no es óptimo en lo que concierne al número de comparaciones. Esta situación no ocurre con el árbol de decisión de la figura 12.4, pero cuidado con las apariencias: ocurre con mucha más frecuencia en el algoritmo de inserción que con *ordenación por montículo* cuando aumenta el número de elementos que hay que ordenar.

Como en el juego de las veinte preguntas, la altura del árbol de decisión correspondiente a cualquier algoritmo para ordenar n elementos por comparación da el número de comparaciones efectuadas por este algoritmo en el caso peor. Por ejem-

plo, se encuentra un posible caso peor para ordenar tres elementos por inserción cuando el vector está ya en orden descendente $C < B < A$; en este caso es preciso hacer las tres comparaciones "¿$B < A$?", "¿$C < A$?" y "¿$C < B$?" que están en el camino que va desde la raíz hasta el veredicto adecuado dentro del árbol de decisión.

Todos los árboles de decisión que hemos visto para ordenar tres elementos son de altura 3. ¿Se puede hallar un árbol de decisión válido para ordenar 3 elementos cuya altura sea menor? De ser así, tendremos un algoritmo *in situ* para ordenar tres elementos que sería más eficiente en el caso peor. Inténtelo: pronto verá que no se puede hacer. La razón es que todo algoritmo correcto de ordenación debe ser capaz de producir al menos seis veredictos diferentes cuando $n = 3$, puesto que este es el número de permutaciones de tres elementos. En el caso peor, resolver el problema de ordenar tres elementos con menos de tres comparaciones es tan imposible —y por la misma razón— como adivinar un entero entre 1 y 6 con menos de 3 preguntas.

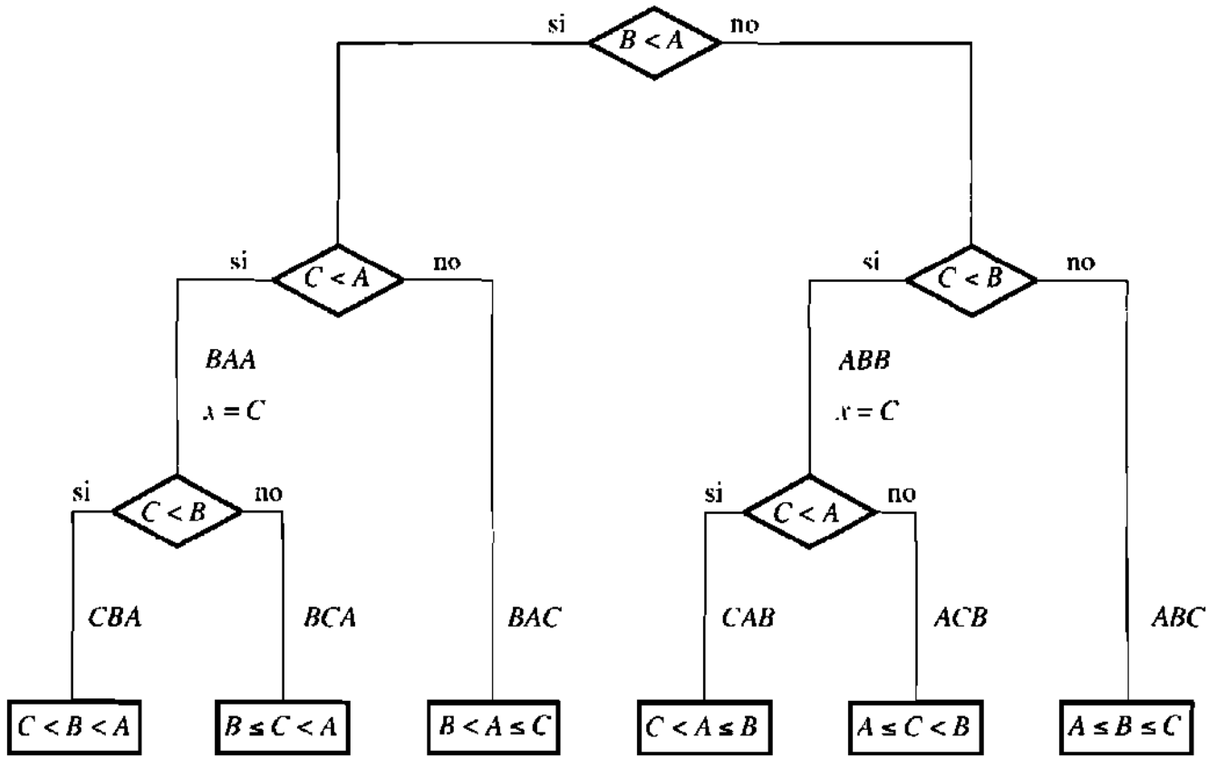


Figura 12.4 El árbol de decisión para la ordenación por inserción de tres elementos

Más generalmente, todo árbol de decisión válido para ordenar n elementos debe contener al menos $n!$ hojas, y por tanto tiene que tener una altura que sea como mínimo $\lceil \lg n! \rceil$ y una altura media que sea como mínimo $\lg n!$ por el problema 12.1 y el Teorema 12.2.1. Esto se traduce directamente en la complejidad de la ordenación: todo algoritmo que ordene n elementos por comparaciones debe hacer al menos $\lceil \lg n! \rceil$ comparaciones en el caso peor, y $\lg n!$ en el caso promedio, siempre y cuando todos los veredictos sean igualmente probables. Dado que toda comparación debe requerir al menos una cantidad constante de tiempo, y puesto que $\lg n! \in \Theta(n \log n)$ por el problema 3.24, se sigue que necesita un tiempo que está en $\Omega(n \log n)$ para ordenar n elementos tanto en el caso peor como en el caso prome-

dio, independientemente de cual sea el algoritmo de ordenación basado en comparaciones que se utilice. De esta manera, vemos que *quicksort* es óptimo en el caso promedio, aun cuando su comportamiento en el caso peor sea lastimoso; véase la Sección 7.4.2.

Hemos demostrado que todo algoritmo determinista para ordenar por comparación debe de hacer al menos $\lceil \lg n! \rceil$ comparaciones en el caso peor cuando se ordenan n elementos. Hay que tener precaución para no hacer que los argumentos de complejidad como este digan cosas que no quieren decir. En particular, el argumento del árbol de decisión *no* implica que siempre sea posible ordenar n elementos con tan solo $\lceil \lg n! \rceil$ comparaciones en el caso peor. de hecho, se ha demostrado que en el caso peor son necesarias y suficientes 30 comparaciones para ordenar 12 elementos, mientras que $\lceil \lg 12! \rceil = 29$.

En el caso peor, el algoritmo de ordenación por inserción hace 66 comparaciones cuando se ordenan 12 elementos, mientras que *ordenación por montículo* hace 59, de las cuales las primeras 18 se hacen durante la construcción del montículo. Por tanto, ambos están lejos de ser óptimos. Sin embargo, se puede demostrar que *ordenar-mont* nunca hace mucho más del doble del número óptimo de comparaciones, mientras que la ordenación por inserción empeora arbitrariamente cuando se hace grande el número de elementos que hay que ordenar. Desde este punto de vista, *ordenar-fusión* es todavía mejor que *ordenación por montículo*, porque hace un número de comparaciones que es casi óptimo; véase el problema 12.7. No vaya a creer, sin embargo, que optimizar el número de comparaciones es una meta en sí: la ordenación por *inserción binaria* hace tan pocas comparaciones como *ordenar-fusión*, y sin embargo requiere una magnitud cuadrática de movimiento de datos—y por tanto de tiempo—en el caso peor. (Este algoritmo es como la ordenación por inserción, salvo que el punto en el cual hay que insertar cada elemento se determina por búsqueda binaria en lugar de hacerlo por búsqueda secuencial).

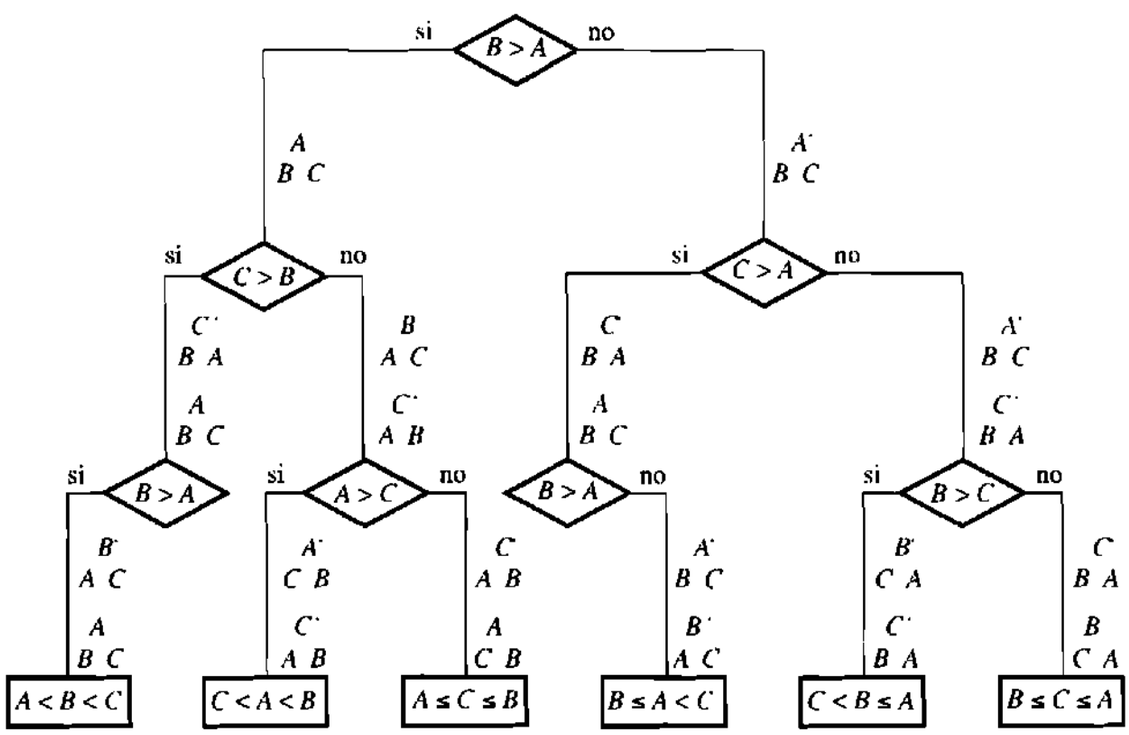


Figura 12.5 El árbol de decisión de heapsort para tres elementos

12.2.2 La complejidad, al rescate de la algoritmia

En algunas ocasiones, las técnicas desarrolladas para la complejidad computacional son útiles para el diseño de algoritmos eficientes. Ilustraremos esto con un rompecabezas clásico. Nos dan 12 monedas y nos dicen que o bien son todas de igual peso o bien 11 son iguales y una es diferente. Nuestra tarea consiste en decidir si todas las monedas son iguales; de no ser así, tenemos que encontrar la moneda diferente y decir si es más ligera o más pesada que las demás. Para hacerlo, nos dan una balanza de platillos. La única operación permitida es colocar unas cuantas monedas en el lado izquierdo de la balanza, otras en el lado derecho y ver si sus pesos son iguales o si el fiel se desvía a uno u otro lado. La figura 12.6 ilustra el pesaje de dos conjuntos de cuatro monedas, observándose que pesa más el lado izquierdo que el lado derecho. Para hacer que el problema tenga interés, nos limitaremos a utilizar la balanza solo tres veces. Le recomendamos que deje el libro e intente hallar su propia solución. Aunque la encuentre, quizá encuentre interesante la discusión que sigue.

Todo algoritmo para resolver este problema se puede representar mediante un árbol de decisión. Cada nodo interno especifica las monedas que están en los platillos de la balanza, y cada hoja da un veredicto tal como "es más pesada la moneda *D*". El árbol debe tener una altura 3 como máximo, puesto que no se nos permite hacer más de tres medidas. ¿A cuántos veredictos tendremos que dar cabida?

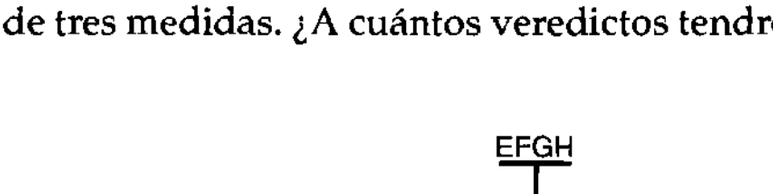


Figura 12.6 Pesaje de ABCD frente a EFGH

Un posible veredicto es que todas las monedas son idénticas; para cada una de las doce monedas, hay también dos veredictos posibles correspondientes a que esa moneda sea bien más ligera, bien más pesada que las demás. El número total de veredictos, por tanto, es $1 + 12 \times 2 = 25$. A primera vista, esto es inquietante, puesto que hemos visto que un árbol de decisión de altura 3 no puede admitir más de $2^3 = 8$ veredictos. Afortunadamente, nuestra árbol de decisión para este problema es ternario, y no binario, porque cada medida tiene tres resultados posibles: el fiel se puede inclinar a la izquierda, puede quedar en equilibrio, o puede inclinarse a la derecha. Del mismo modo que los árboles binarios de altura k tienen un máximo de 2^k hojas, los árboles ternarios de altura k pueden tener un máximo de 3^k hojas. Todo va bien, puesto que un árbol ternario de altura 3 puede admitir hasta $3^3 = 27$ veredictos, y solamente necesitamos 25. Sin embargo, esto no demuestra que el problema tenga solución: recuerde que se necesitan 30 comparaciones en el caso peor para ordenar 12 elementos, aun cuando $\lceil \lg 12! \rceil = 29$. Véase también el problema 12.12.

Los árboles de decisión resultan útiles para hallar la solución, porque nos ayudan a evitar salidas en falso. La idea principal es que tiene que quedar un máximo de $3^2 = 9$ veredictos posibles para cada uno de los tres resultados posibles de la primera medida. En caso contrario, no tendríamos ninguna esperanza de resolver el problema en el caso peor con solo dos medidas más. Obsérvese también que no tiene sentido emplear la balanza con un número de monedas distinto en cada platillo: no se puede extraer información útil si el fiel se inclina hacia el lado que contenga más monedas. Por tanto, primero utilizamos la balanza para comparar dos conjuntos de k monedas, para algún $1 \leq k \leq 6$. Si el fiel queda en equilibrio, la moneda diferente puede ser una de las $12-2k$ monedas restantes, lo cual nos deja $25-4k$ veredictos posibles, contando con la posibilidad de que todas las monedas puedan ser iguales. Si el fiel se inclina a un lado, por otra parte, puede ser o bien porque una de las monedas de ese lado es más pesada, o bien porque una de las monedas del otro lado es más ligera, lo cual nos deja con $2k$ veredictos posibles. Tal como hemos visto, no hay esperanza de poder resolver el problema si después de la primera medida quedan más de 9 veredictos posibles. Por tanto, necesitamos simultáneamente $25 - 4k \leq 9$ y $2k \leq 9$. La única solución entera de estas ecuaciones es $k=4$. Este razonamiento sigue sin demostrar que se pueda hallar una solución si empezamos por comparar dos conjuntos de cuatro monedas, pero ciertamente nos dice que no tiene sentido intentar alguna otra cosa.

Después de la primera medida, hay que considerar dos casos: o bien uno de los conjuntos de monedas pesa más que el otro, o bien los dos pesan lo mismo. En ambos casos, la segunda medida tiene que ser tal que una vez conocido su resultado solo sean posibles tres veredictos como máximo. Esto se debe a que la última medida no puede distinguir entre más de tres posibilidades. Si en la primera medida se encuentra que uno de los conjuntos pesa más que el otro, un razonamiento similar al anterior muestra que la segunda medida debe de implicar o bien 5 o bien 6 de las 8 monedas empleadas en la primera medida. Sabido esto, es fácil completar los detalles y determinar exactamente las dos últimas medidas.

La situación es mucho más interesante si en la primera medida se obtiene que ambos conjuntos pesan lo mismo: parece que hemos vuelto a lo que se parece al problema original, salvo que tenemos cuatro monedas en lugar de doce, y solo se nos permiten dos medidas en lugar de tres. Empleando una vez más argumentos de teoría de la información, no se tarda mucho en averiguar que esta versión reducida del problema no tiene solución (problema 12.13), y a primera vista que esto implica también el fin del problema original. Lo que nos salva es que se nos permite utilizar algunas de las ocho monedas que participaron en la primera medida, aun cuando sepamos que la moneda que buscamos no se encuentra entre ellas. Dado que los argumentos de teoría de la información nos dicen que no podemos tener éxito si no utilizamos al menos una de las monedas iniciales, sabemos que debemos hacer esto si queremos llegar a tener éxito. En este momento, no queda gran cosa por averiguar, y le invitamos a que haga el problema 12.11 para los detalles.

12.3 ARGUMENTOS DEL ADVERSARIO

Dado un problema, deseamos demostrar la validez de una cota inferior para el tiempo requerido en el caso peor por parte de cualquier algoritmo que lo resuelva correctamente en todos los casos. A lo largo de toda esta sección, asumiremos por sencillez que los algoritmos son deterministas, aun cuando los algoritmos probabilistas se pueden considerar también, empleando argumentos más sutiles. La idea que subyace a los argumentos del adversario es poner en marcha el algoritmo con una entrada que inicialmente no está especificada, salvo por su tamaño. Cuando el algoritmo examina la entrada, un diablillo malevolente, conocido como el adversario, responde de tal manera que obliga al algoritmo a trabajar mucho. El diablillo tiene que ser consecuente, en el sentido de que siempre tiene que existir al menos una entrada que haría que el algoritmo viera exactamente las respuestas del diablillo a cada una de sus preguntas. El objetivo del diablillo es mantener al algoritmo incierto sobre la respuesta correcta durante el mayor tiempo posible. Si el algoritmo afirma conocer la respuesta antes de haber examinado la entrada un número suficiente de veces, el diablillo debe mostrar una posible entrada que sea consistente con todas sus respuestas a las preguntas del algoritmo, pero cuya solución correcta sea distinta a la salida del algoritmo. Dado que esta podría haber sido la entrada real del algoritmo, este último puede equivocarse a no ser que examine suficientemente la entrada en el caso peor.

Considere una vez más el juego de las veinte preguntas presentado en la Sección 12.1. La forma honesta de jugar por parte de su amiga es seleccionar un número entre 1 y un millón, incluso antes de que formule usted la primera pregunta. Si ella es falaz, sin embargo, puede retrasar su elección mientras sea posible, haciendo que dependa de las preguntas que usted le haga. Su objetivo es responder a las preguntas de forma consistente con las respuestas anteriores, pero también de tal manera que le obligue a hacer el mayor número posible de preguntas. Cuando se hace la pregunta Q_i, ella forma los conjuntos N_i y Y_i descritos en la Sección 12.1, y responde "si" si y solo si Y_i contiene más elementos que N_i. Está obligada a decidir su número misterioso solo cuando quede un único candidato. De esta manera, sea cual sea la estrategia utilizada, su amiga se asegura de que con cada pregunta el número de candidatos se divida por dos en el mejor de los casos. Si usted afirma conocer la respuesta al cabo de menos de 20 preguntas, queda más de un candidato, así que hay al menos un número distinto de su selección, que es consistente con toda la información de la que dispone. Su amiga podría entonces decirle que se ha "equivocado" fingiendo que ha seleccionado su número desde un principio de tal modo que fuera uno de estos.

Una vez que se han comprendido los argumentos de teoría de la información, son más fáciles de utilizar que los argumentos del adversario para demostrar que se requieren 20 preguntas para jugar a este juego en el caso peor. Sin embargo, hay problemas para los cuales los argumentos de la teoría de la información resultan inútiles, pero se pueden emplear los argumentos del adversario. Ahora daremos tres ejemplos; los dos primeros son elementales, mientras que el tercero es más complicado.

12.3.1 Búsqueda del máximo en un vector

Considere un vector $T[1..n]$ de enteros. Suponemos que $n > 0$. Nuestra tarea consiste en hallar el índice de un valor máximo alcanzado por el vector. Por ejemplo, si $n = 7$ y $t = [2, 7, 1, 8, 2, 8, 1]$, tanto 4 como 6 son respuestas correctas, puesto que $T[4] = T[6] = 8$ es el mayor valor del vector. Como en la ordenación basada en comparaciones (Sección 12.2.1), la única operación permitida sobre estos elementos es compararlos por parejas. El algoritmo evidente efectúa exactamente n–1 comparaciones.

```
función indicemax(T[1..n])
    m ← T[1]; k ← 1
    para i ← 2 hasta n hacer
        si T[i] > m entonces m ← T[i]
                             k ← i
    devolver k
```

¿Es posible hacerlo mejor?

Si intentamos emplear un argumento de teoría de la información, encontraremos que todo árbol de decisión para este árbol debe contener n veredictos posibles. Dado que el árbol es binario, tiene que tener una altura que sea como mínimo $\lceil \lg n \rceil$. Por tanto, todo algoritmo basado en comparaciones para hallar el máximo debe de efectuar al menos $\lceil \lg n \rceil$ comparaciones en el caso peor. Esto está muy lejos de n–1, que es lo mejor que sabemos hacer. ¿Podrán obtener los argumentos adversarios una cota inferior más ajustada?

Considere un algoritmo arbitrario basado en comparaciones para este problema. Apliquémoslo a un vector $T[1..n]$, sin especificar por el momento. El diablillo responde a todas la preguntas concernientes a una comparación entre elementos como si $T[i]$ fuera igual a i para todo i. Cada vez que el algoritmo pregunta "¿Es $T[i] < T[j]$?" para $i \neq j$, decimos que el menor de i y j ha "perdido en una comparación". Supongamos que el algoritmo efectúa menos de n–1 comparaciones antes de emitir la respuesta k. Sea j un entero distinto de k, $1 \leq j \leq n$, que no haya perdido en ninguna comparación. Este j existe porque por hipótesis se han efectuado como máximo n–2 comparaciones, y cada comparación crea, como máximo, un nuevo perdedor. Llegados aquí, el diablillo puede afirmar que el algoritmo ha fallado. Para hacer esto, fingirá que era $T[i] = i$ para todo $i \neq j$, pero que $T[j] = n+1$. En efecto, $T[k] = k$ no es el máximo de este vector, y sin embargo las respuestas obtenidas por este algoritmo son consistentes con esto. Esto completa la demostración de que se necesitan n–1 comparaciones para resolver el problema del máximo mediante cualquier algoritmo basado en comparaciones.

Este resultado no es válido si además de las comparaciones se admite la aritmética aplicada a los elementos. Para hacerlo mejor, calcule $a = \sum_{i=1}^{\ell} n^{T[i]}$ y $b = \sum_{i=\ell+1}^{n} n^{T[i]}$, donde $\ell = \lceil n / 2 \rceil$. Si existe un elemento x dentro de $T[1..\ell]$ que sea mayor que cualquier elemento de $T[\ell + 1..n]$, entonces $a > b$ porque

$$\sum_{i=1}^{\ell} n^{T[i]} \geq n^{\lambda} > \tfrac{1}{2} n \times n^{\lambda-1} \geq \sum_{i=\ell+1}^{n} n^{T[i]}.$$

De manera similar, $a < b$ si el máximo de T se encuentra en la segunda mitad. Si el máximo aparece en ambas mitades, puede darse cualquier relación entre a y b. De esta forma, una sola comparación entre a y b basta para determinar si el máximo se encuentra en la primera mitad de t o en la segunda. Prosiguiendo igual que en una búsqueda binaria, podemos hallar la respuesta al cabo de $\lceil \lg n \rceil$ comparaciones como máximo. Por supuesto, sería una tontería emplear esta aproximación en la práctica, puesto que cambia unas comparaciones de bajo coste por un elevado número de operaciones aritméticas que consumen mucho tiempo.

12.3.2 Comprobación de la conectividad de un grafo

Considere un algoritmo que comprueba si un grafo no dirigido con n vértices, $n \geq 2$, es o no conexo. Las únicas preguntas que se admiten son de la forma "¿Existe una arista entre los vértices i y j?" Sabemos por la Sección 9.3 que este problema se puede resolver mediante un recorrido en profundidad, que requiere un tiempo en $O(n^2)$ en el caso peor. ¿Será posible resolverlo más rápidamente empleando un algoritmo más sofisticado? Los argumentos de teoría de la información son inútiles con este problema, porque solo hay dos veredictos posibles: o bien el grafo es conexo, o bien no lo es. Por tanto el árbol de decisión tiene un altura mínima de 1, lo cual quiere decir que el algoritmo tiene que hacer al menos una pregunta. ¡Buena respuesta!

Para diseñar un argumento del adversario, el diablillo descompone el conjunto de vértices en dos subconjuntos V y W de tamaños $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil$ respectivamente. Cada vez que el algoritmo pregunta si hay o no una arista entre los vértices i y j, el diablillo dice "si" si y solo si ambos vértices pertenecen a el mismo subconjunto. Si el algoritmo afirma que conoce la respuesta antes de preguntar acerca de la existencia de todas las aristas potenciales cuyos extremos no se encuentran en un mismo subconjunto, el diablillo puede demostrar que se equivoca en la forma siguiente. Si el algoritmo afirma que el grafo es conexo, el diablillo muestra el grafo inconexo que posee una arista entre los vértices i y j si y solo si estos vértices se encuentran en el mismo subconjunto. Este grafo no es conexo, porque no hay caminos entre los vértices de V y los vértices de W. Por otra parte, si el algoritmo afirma que el grafo no es conexo, entonces el diablillo muestra el grafo conexo que se obtiene de la misma manera, pero añadiendo una arista adicional entre V y W que no haya sido consultado por el algoritmo. Esta arista existe por la suposición de que el algoritmo no ha consultado todas las posibles aristas de este tipo. Claramente, el diablillo produce un grafo consistente con las consultas del algoritmo, grafo que será conexo si y solo si el algoritmo decía que no lo es. Esto demuestra que todo algoritmo que no haga por lo menos $\lfloor n/2 \rfloor \times \lceil n/2 \rceil \in \Omega(n^2)$ preguntas en el caso peor no puede ser correcto. Por tanto, el recorrido en profundidad es óptima en el caso peor, salvo la constante multiplicativa oculta.

No deduzca de esto que basta con hacer $\lfloor n/2 \rfloor \times \lceil n/2 \rceil$ preguntas para resolver el problema en general. Una dificultad es que el algoritmo no sabe que estrategia va a utilizar el diablillo, e incluso si lo supiera, no conocería la forma en que se ha

repartido el conjunto de vértices entre V y W. De hecho, se puede demostrar mediante un argumento del adversario más sofisticado que en el caso peor todo algoritmo correcto tiene que consultar todas y cada una de las $n(n-1)/2$ posibles aristas; véase el problema 12.16.

12.3.3 La mediana, segunda parte

Vimos en la Sección 7.5 que basta un número de comparaciones en $O(n)$ para hallar la mediana de un vector $T[1..n]$ formado por enteros. Claramente, también se necesita un tiempo en $\Omega(n)$ porque cuando todos los elementos son distintos debemos examinarlos todos al menos una vez. Con cualquier algoritmo basado en comparaciones, la cota inferior trivial sobre el número exacto de comparaciones es por tanto $\lceil n/2 \rceil$, porque cada comparación examina solamente dos elementos y es preciso examinarlos todos. ¿Se puede hallar una cota inferior más interesante?

Supongamos por sencillez que n es impar y $n \geq 3$. Demostraremos que cualquier algoritmo basado en comparaciones necesita un mínimo de $3(n-1)/2$ comparaciones para localizar la mediana en el caso peor. Si el algoritmo no hace todas estas comparaciones, el diablillo puede obligarle a dar una respuesta incorrecta. Sin pérdida de generalidad, suponemos que el algoritmo nunca compara a un elemento consigo mismo; si lo hiciera, el diablillo respondería con la única respuesta razonable. Suponemos también que todo elemento de T aparece al menos en una comparación solicitada por el algoritmo. En caso contrario, es imposible que el algoritmo localice la mediana con certeza, y la tarea del diablillo sería sencilla. Esto es cierto porque veremos que el diablillo nunca da a dos elementos de T el mismo valor.

Inicialmente, el diablillo da a todas las entradas de T el valor "no inicializado". A medida que el algoritmo hace comparaciones, el diablillo cambia estos valores. Algunos elementos de T reciben valores entre 1 y n, mientras que otros reciben valores entre $3n + 1$ y $4n$. Llamaremos a estos valores "bajos" y "altos" respectivamente. El diablillo se asegura de que siempre haya tantos valores altos como bajos. Intuitivamente, los valores bajos serán más pequeños que la mediana, y los valores altos serán mayores. Sin embargo, el diablillo puede cambiar los valores de T siempre que lo desee, con tal de que las respuestas que diera previamente al algoritmo sigan siendo válidas. Por tanto, uno de los valores que anteriormente era bajo o alto puede acabar por ser finalmente la mediana. Cada vez que el algoritmo pide una comparación entre $T[i]$ y $T[j]$, existen tres posibilidades.

- ◊ Si tanto $T[i]$ como $T[j]$ no han sido inicializados, el diablillo da a $T[i]$ el valor i y a $T[j]$ el valor $3n + j$. De esta forma, $T[i]$ pasa a ser bajo y $T[j]$ pasa a ser alto.
- ◊ Si sólo ha sido iniciado o bien $T[i]$ o bien $T[j]$, entonces hay cinco subcasos.
  - Si queda sin iniciar un solo elemento de T, el diablillo le da el valor $2n$, que no es ni alto ni bajo. Este elemento pasa a ser la *mediana provisional*: todos los elementos bajos son más pequeños, todos los elementos altos son más grandes, y hay tantos elementos bajos como elementos altos. Sin embargo, la mediana puede llegar a estar finalmente en algún otro lugar.

- En caso contrario, si $T[i]$ es bajo, el diablillo da a $T[j]$ el valor alto $3n + j$. Para restaurar el equilibrio entre elementos bajos y altos, selecciona un $T[k]$ arbitrario no inicializado y le da el valor bajo k.
- Si $T[j]$ es bajo, el diablillo actúa como en el subcaso anterior, intercambiando i y j.
- Si $T[i]$ es alto, el diablillo da a $T[j]$ el valor bajo j. Para restaurar el equilibrio entre elementos bajos y altos, selecciona un $T[k]$ arbitrario no inicializado y le da el valor alto $3n + k$.
- Si $T[j]$ es alto, el diablillo actúa como en el subcaso anterior, intercambiando i y j.

◊ En caso contrario, tanto $T[i]$ como $T[j]$ ya han sido inicializados. Si ambos son bajos o si uno es bajo y el otro es la mediana provisional, diremos que el menor "ha perdido en una comparación". Ninguno pierde en la comparación si uno es bajo y el otro es alto.

Finalmente, el diablillo responde a la solicitud de una comparación por parte del algoritmo de acuerdo con los valores actuales de $T[i]$ y $T[j]$.

Supongamos ahora que el algoritmo hace menos de $3(n–1)/2$ comparaciones antes de devolver su valor para la mediana. Dado que los elementos de T se van iniciando por parejas, salvo por la mediana provisional, y teniendo en cuenta que hemos supuesto que todos los elementos aparecen al menos en una comparación, exactamente $(n–1)/2$ comparaciones habrán sido bien del primer o del segundo tipo mencionados más arriba. Consiguientemente, menos de $\frac{3}{2}(n-1) - \frac{1}{2}(n-1) = n-1$ comparaciones implicaban a dos elementos que ya estuvieran iniciados. Dado que los elementos solo pueden perder una comparación en este caso, al menos un elemento además de la mediana provisional no habrá perdido nunca una comparación. Llamémoslo $T[i]$. Supongamos sin pérdida de generalidad que es bajo. Por la definición de pérdida de comparación, el diablillo nunca había tenido que admitir al algoritmo que este elemento es menor que cualquier otro elemento bajo, o menor que la mediana provisional. Por tanto, no aparece una contradicción si el diablillo cambia de opinión e incrementa $T[i]$, siempre y cuando lo haga menor que cualquier elemento alto. Esto da al demonio la opción de mantener $T[i] = i$, de tal forma que la mediana provisional sea realmente la mediana final, o bien de hacer $T[i] = 2n + i$, para que $T[i]$ pase a ser la mediana final. Sea cual fuere la respuesta devuelta por el algoritmo, el diablillo puede mostrar un vector T con una mediana distinta, sin dejar de ser consistente con todas las respuestas vistas por el algoritmo. Esto demuestra que el algoritmo es incorrecto a no ser que haga al menos $3(n–1)/2$ comparaciones en el caso peor.

12.4 REDUCCIONES LINEALES

Ya vimos que todo algoritmo para ordenar por comparación requiere un tiempo mínimo que está en $\Omega(n \log n)$ para ordenar n elementos, tanto por término medio como en el caso peor. Por otra parte, sabemos que *ordenar-mont* y *ordenar-fusión* resuelven ambos el problema en un tiempo que está en $O(n \log n)$. Salvo por el valor de la constante multiplicativa, la pregunta concerniente a la complejidad

de una ordenación por comparación queda zanjada: un tiempo en $\Theta(n \log n)$ es tanto necesario como suficiente para ordenar n elementos. Desafortunadamente, en nuestro presente estado de ignorancia no es frecuente que las cotas derivadas de la algoritmia y de la complejidad coincidan de manera tan satisfactoria.

Dado que es tan difícil determinar la complejidad exacta de la mayoría de los problemas que encontramos en la práctica, es frecuente que tengamos que contentarnos con comparar las dificultades relativas de problemas diferentes. Diremos que un problema *se reduce* a otro si podemos transformar eficientemente casos del primer problema en casos del segundo, de tal manera que la resolución del caso transformado proporcione la respuesta del caso original. Supóngase que demostramos que un cierto número de problemas son equivalentes, en el sentido de que unos se reducen a otros. Por consiguiente, todos estos problemas tienen aproximadamente la misma complejidad: cualquier mejora algorítmica del método de resolución de uno de ellos da lugar automáticamente, al menos en teoría, a un algoritmo más eficiente para todos los demás. Desde un punto de vista negativo, si todos estos problemas se han estudiado independientemente en el pasado, y si no han tenido éxito todos los esfuerzos para hallar un algoritmo eficiente para cualquiera de ellos, entonces el hecho de que estos problemas sean equivalentes hace aún más improbable el que exista ese algoritmo. La Sección 12.5 estudia con detalle esta segunda motivación.

Ya hemos encontrado un ejemplo de reducción: en la Sección 10.7.4 veíamos que la factorización se reduce a la descomposición y a la comprobación de primalidad. De manera similar, tanto la descomposición como la comprobación de primalidad se reducen evidentemente a la factorización, porque ambas quedan resueltas inmediatamente una vez que el número considerado se ha factorizado en forma de un producto de números primos. Aun cuando no conocemos la complejidad exacta de estos problemas, esto dice que la factorización es de dificultad similar a las tareas combinadas de descomposición y comprobación de primalidad.

Otro ejemplo sencillo es el que concierne a la multiplicación y elevación al cuadrado de números enteros muy grandes. Veíamos en la Sección 7.1 que es posible multiplicar dos números de n cifras en un tiempo que está en $O(n^{1.59})$, lo cual es una mejora con respecto al algoritmo clásico, que es de tiempo cuadrático. ¿Se puede hacer mejor? Veíamos en los problemas 7.2 y 7.3 que la respuesta es "si": se puede multiplicar en un tiempo que está en $O(n^{\alpha})$, para cualquier número real $a > 1$, aunque la constante multiplicativa oculta va creciendo a medida que a se hace más pequeño. Incluso mejor: Schönhage y Strassen han descubierto un algoritmo de multiplicación que requiere un tiempo que está en $O(n \log n \log n \log n)$. ¿Es este el mejor posible? La búsqueda de un algoritmo mejor es una cuestión algorítmica; de demostrar que ese algoritmo es óptimo concierne a la complejidad. Desafortunadamente, no se conoce ninguna cota inferior no trivial para la complejidad de la multiplicación. La multiplicación de dos números de n cifras requiere evidentemente un tiempo que está en $\Omega(n)$, porque es preciso examinar todos y cada uno de los dígitos de los operandos. Por lo que nosotros sabemos, podría existir un algoritmo de multiplicación que aún no se haya descubierto y que satisfaga esta cota, efectuando las multiplicaciones en un tiempo que esté en $O(n)$.

Consideremos ahora el problema, aparentemente más sencillo, de elevar al cuadrado números muy grandes. ¿Será posible abordar *este* problema en tiempo lineal, aunque no sea posible hacerlo para la multiplicación? Aun cuando la complejidad exacta de la elevación al cuadrado es tan elusiva como la complejidad de la multiplicación, hay una cosa que si sabemos demostrar: salvo por una pequeña constante multiplicativa, ambas requieren el mismo tiempo. Por tanto, se puede resolver una de ellas en un tiempo lineal si y solo si es posible hacerlo con ambas. Hay algo más interesante, y es que cualquier mejora significativa del arte de elevar al cuadrado produciría un mejor algoritmo de multiplicación. Intuitivamente, esto se sigue de dos fórmulas que relacionan los dos problemas entre sí.

$$x^2 = x \times x$$

$$x \times y = \frac{(x+y)^2 - (x-y)^2}{4}$$

A partir de estas fórmulas se ve que una elevación al cuadrado no puede ser más difícil que una multiplicación, mientras que una multiplicación no puede ser mucho más difícil que dos elevaciones al cuadrado.

Formalmente, demostramos la equivalencia computacional de estos problemas exhibiendo los dos algoritmos. Cada uno de ellos resuelve un problema invocando a un algoritmo arbitrario para el otro. Estos algoritmos se podrían emplear para enseñar una operación a una persona que solo supiera realizar la otra.

```
función cuadrado(x)
   devolver mult(x, x)

función mult(x,y)
   devolver (cuadrado(x+y) – cuadrado(x–y))/4
```

Sean $M(n)$ y $S(n)$ los tiempos necesarios para multiplicar y elevar al cuadrado enteros de tamaño n como máximo, respectivamente. El primer algoritmo deja claro que $S(n) \leq M(n) + c$, para una pequeña constante c que tiene en cuenta el tiempo adicional de *cuadrado* más allá del tiempo invertido en *mult*. Por tanto, $S(n) \in O(M(n))$.

Es preciso analizar el segundo algoritmo con más cuidado, porque $x + y$ puede tener una cifra más que x e y. Supongamos, sin pérdida de generalidad, que $x \geq y$ para no tener que preocuparnos por la posibilidad de que $x - y$ pudiera ser negativo. De esta manera

$$M(n) \leq S(n+1) + S(n) + f(n) \qquad (12.1)$$

en donde $f(n)$ es el tiempo que se necesita para realizar la suma, la resta y la división por 4 que requiere el algoritmo, además del coste general de *mult*. Ya sa-

bemos que las sumas y las restas se pueden efectuar en un tiempo lineal. La división por 4 es trivial si se representan los números en binario, y se puede hacer en tiempo lineal aun cuando se utilice otra base; véase el problema 12.21. Por tanto, $f(n) \in O(n)$ es despreciable puesto que tanto $M(n)$ como $S(n)$ están en $\Omega(n)$. Sin embargo, la Ecuación 12.1 no basta para concluir que $M(n) \in O(S(n))$. Considere lo que pasaría si el algoritmo de elevación al cuadrado fuera tan inepto que requiriese un tiempo en $\Theta(n!)$ para elevar al cuadrado un número de n cifras. En tal caso, la multiplicación de dos números de este tamaño podría requerir un tiempo que estuviese en $\Theta((n+1)!)$, que es aproximadamente n veces mayor que el tiempo invertido en elevar al cuadrado un número así. Sin embargo, $M(n) \in O(S(n))$ si que se sigue de una suposición razonable; véase el Teorema 12.4.4 más adelante. Entonces concluimos que los problemas de multiplicación y elevación al cuadrado de números enteros tiene la misma complejidad, salvo por una constante multiplicativa.

¿Qué se puede decir acerca de otras operaciones aritméticas elementales, tales como la división entera y la extracción de raíces cuadradas? La experiencia de todos los días nos lleva a creer que el segundo de estos problemas, y quizá también el primero, son verdaderamente más difíciles que la multiplicación. Esto resulta no ser cierto. Con suposiciones razonables, se necesita el mismo tiempo para multiplicar dos números de n cifras, para calcular el cociente cuando se divide un número de $2n$ cifras por otro de n cifras, y para calcular la parte entera de la raíz cuadrada de un número de n cifras. Eso se debe a fórmulas tan exóticas como

$$x^2 = \frac{1}{\frac{1}{x} - \frac{1}{x+1}} - x$$

$$\frac{1}{x} \approx \sqrt{x + \sqrt{x+1}} - \sqrt{x-1} - \sqrt{x - \sqrt{x+1} + \sqrt{x-1}}$$

y técnicas clásicas tales como el método de Newton para hallar los ceros de las funciones. Haga los problemas 12.27 y 12.28 para más detalles.

Además de su utilidad para propósitos de complejidad, el hecho de que una multiplicación se reduzca a dos elevaciones al cuadrado es interesante desde el punto de vista algorítmico, ofrece un ejemplo instructivo de precondicionamiento; véase la Sección 9.2. Suponga por un momento que fuera usted romano, y que no dispusiera de otra notación para los números. Si tuviese que multiplicar frecuentemente los números del 1 al 1000, quizá mereciese la pena compilar una tabla de multiplicar, de una vez por todas. Sin embargo, la tabla abarcaría más de medio millón de entradas, aun cuando solo se calcule el producto de x por y cuando fuera $x \geq y$. Una solución mejor sería compilar una tabla de valores para $x^2/4$ para todo x entre 1 y 2000. (Tendrá que inventar un símbolo pseudoromano para denotar un cuarto). Una vez tenida esta tabla, se puede efectuar cualquier multiplicación requerida haciendo una suma y dos restas, junto con dos búsquedas en la tabla.

12.4.1 Definiciones formales

Definición 12.4.1. *Sean* **A** *y* **B** *dos problemas. Diremos que* **A** *es linealmente reducible a* **B**, *se denota* $\mathbf{A} \leq' \mathbf{B}$, *si la existencia de un algoritmo para* **B** *que funcione en un tiempo en* $O(t(n))$ *para una función arbitraria* $t(n)$ *implica que existe un algoritmo para* **A** *que también funciona en un tiempo en* $O(t(n))$. *Cuando se tiene a la vez que es* $\mathbf{A} \leq' \mathbf{B}$ y $\mathbf{B} \leq' \mathbf{A}$, *diremos que* **A** *y* **B** *son linealmente equivalentes, se denota* $\mathbf{A} \equiv' \mathbf{B}$.

Intuitivamente, $\mathbf{A} \leq' \mathbf{B}$ significa que a alguien que sepa hacer el problema *B* se le puede enseñar a hacer también el problema *A*. Se sigue del problema 3.10 que las reducciones lineales son transitivas: si $\mathbf{A} \leq' \mathbf{B}$ y $\mathbf{B} \leq' \mathbf{C}$, entonces $\mathbf{A} \leq' \mathbf{C}$. Las equivalencias lineales también son transitivas, además de ser reflexivas y simétricas.

Formalmente, demostraremos que $\mathbf{A} \leq' \mathbf{B}$ exhibiendo un algoritmo que resuelva un caso de **A** transformándolo en uno o más casos del problema **B**. La conclusión de que $\mathbf{A} \leq' \mathbf{B}$ se sigue inmediatamente si los casos requeridos para el problema **B** son del mismo tamaño que el caso original del problema **A**, si se necesita un número constante de ellos, y si la cantidad de trabajo adicional a la resolución de esos casos no es mayor (salvo una constante multiplicativa) que el tiempo requerido para resolver el problema **B**. Por ejemplo, vimos que elevar un entero al cuadrado se reduce a una sola multiplicación del mismo tamaño más una cantidad constante de trabajo. Por tanto, la elevación de enteros al cuadrado se reduce linealmente a la multiplicación de enteros.

La situación se complica más cuando la resolución de casos de tamaño n del problema **A** implica casos del problema **B** que tienen distinto tamaño, o cuando implica un número de casos del problema **B** que no está acotado por una constante. Por ejemplo, multiplicar dos enteros se reduce a elevar dos enteros al cuadrado, de los cuales uno tiene que ser de tamaño ligeramente superior. Para manejar con precisión estas situaciones, tenemos que derivar una expresión para el tiempo requerido por el algoritmo de reducción para resolver un caso de tamaño n del problema **A**, como función del tiempo requerido para resolver el problema **B** en casos de tamaño arbitrario. La Ecuación 12.1 es típica de este enfoque. Para concluir la relación deseada entre los tiempos que se necesitan para resolver ambos problemas, sin embargo, suele ser necesario hacer suposiciones relacionados con el concepto de suavidad que presentábamos en la Sección 3.4, y otros conceptos que se presentan más adelante.

Definición 12.4.2. *Un* algoritmo *es* suave *si requiere un tiempo que está en* $\Theta(t(n))$ *para alguna función suave* $t(n)$. *Un* problema *es suave si todo algoritmo razonable que lo resuelva es suave. (Al decir "razonable" queremos decir que el algoritmo no es deliberadamente más lento que lo necesario; véase más adelante).*

Recuerde que la función $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ es suave si es eventualmente no decreciente y si $f(bn) \in O(f(n))$ para todo entero $b \geq 2$. Extendemos esta definición a los algoritmos y a los problemas.

Aun cuando una función suave debe de ser asintóticamente no decreciente por definición, esto no implica que el tiempo real que requiera una implementación específica de un algoritmo suave deba estar dado también por una función asintóticamente no decreciente. Considere por ejemplo el algoritmo de exponenciación modular *expomod* de la Sección 7.8. Con este algoritmo se tarda más en calcular una potencia $(2^{k+1}-1)$-ésima que una potencia 2^k-ésima para cualquier k grande, y por tanto el tiempo real requerido por cualquier implementación razonable de este algoritmo *no* es una función asintóticamente no decreciente del exponente. Sin embargo, este algoritmo *es* suave porque necesita un tiempo que está en $\Theta(\log n)$ para calcular una potencia n–ésima, contando las multiplicaciones con coste unitario, y $\log n$ es una función suave. La restricción de que el algoritmo sea razonable es necesaria en la definición de un problema suave, porque todo problema que sea posible resolver de algún modo se podrá resolver mediante un algoritmo que requiera un tiempo en $\Omega\, 2^n$ —el algoritmo comienza por un bucle inútil que cuenta desde 1 hasta 2^n— y ese algoritmo no puede ser suave; véase el problema 12.22. Otro ejemplo de algoritmo que no es razonable sería uno que ordenase n elementos probando sistemáticamente las $n!$ permutaciones hasta hallar una que estuviera ordenada: esto no es razonable porque existen algoritmos de ordenación que son mucho más eficientes.

Definición 12.4.3. *Una función* $t: \mathbb{N} \to \mathbb{R}^{\geq 0}$ al menos cuadrática *si* $t(n) \in \Omega(n^2)$. *La función se denominará* fuertemente cuadrática *si es eventualmente no decreciente y* $t(an) \geq a^2 t(n)$ *para todo entero positivo* a *y para todo entero* n *suficientemente grande. Las funciones* al menos lineales *y* fuertemente lineales *se definen de forma similar. Estas nociones se extienden a algoritmos y problemas del mismo modo que las funciones suaves.*

Es sencillo demostrar que las funciones fuertemente cuadráticas son al menos cuadráticas. La mayoría de los teoremas siguientes se enuncia condicionalmente con una suposición "razonable", como $\mathbf{A} \leq \mathbf{B}$ suponiendo que **B** sea suave. Esto se puede interpretar literalmente, como indicación de que $\mathbf{A} \leq \mathbf{B}$ se sigue de la suposición de que **B** sea suave. Desde un punto de vista más práctico, también significa que para cualquier función suave $t(n)$, la existencia de un algoritmo para **B** que funcione en un tiempo que esté en $O(t(n))$ implica que existe un algoritmo para **A** que también funciona en un tiempo que está en $O(t(n))$. Además, todos estos teoremas son constructivos; el algoritmo para **A** se sigue del algoritmo para **B** y de la demostración del teorema correspondiente.

Ahora estamos en situación adecuada para enunciar con precisión y demostrar la equivalencia lineal entre la elevación de enteros al cuadrado y la multiplicación de enteros.

Teorema 12.4.4. *Sean* SQR *y* MLT *los problemas consistentes en elevar al cuadrado un entero de tamaño n y multiplicar dos enteros de tamaño n, respectivamente. Suponiendo que* SQR *sea suave, los dos problemas son linealmente equivalentes.*

Demostración: Vimos anteriormente que ambos problemas tienen que ser al menos lineales. Sean $M(n)$ y $S(n)$ los tiempos necesarios, respectivamente, para multiplicar y elevar al cuadrado operandos de tamaño n como máximo. Vimos que existe una constante c y una función $f(n) \in O(n)$ tales que $S(n) \leq M(n) + c$, y $M(n) \leq S(n+1) + S(n) + f(n)$. Dado que $M(n)$ es al menos lineal, $M(n) \geq c$ para todo n suficientemente grande. Por tanto, $S(n) \leq 2M(n)$, también para todo n suficientemente grande, lo cual implica por definición que $S(n) \in O(M(n))$ y por tanto que $SQR \leq {}^{t}MLT$.

Recíprocamente, suponga que $S(n) \in \Theta(s(n))$ para alguna función suave $s(n)$. Sean a, b_1 y b_2 unas constantes adecuadas, tales que $s(2n) \leq as(n)$, $S(n) \leq b_1 s(n)$ y $s(n) \leq b_2 S(n)$ para todo n suficientemente grande. Dado que toda función suave es asintóticamente no decreciente por definición,

$$S(n+1) \leq b_1 s(n+1) \leq b_1 s(2n) \leq b_1 as(n) \leq b_1 ab_2 S(n)$$

para todo n suficientemente grande. Dado que $f(n) \in O(n)$ y que $S(n) \in \Omega(n)$, existe una constante d tal que $f(n) \leq dS(n)$ para todo n suficientemente grande. Reuniendo todo esto, concluimos que

$$M(n) \leq S(n+1) + S(n) + f(n) \leq (b_1 ab_2 + 1 + d)S(n)$$

para todo n suficientemente grande, y por tanto $M(n) \in O(S(n))$ y $MTL \leq {}^{t}SQR$ por definición.

Daremos otros ejemplos de reducciones lineales más adelante, y sugerimos otros en los ejercicios. En la Sección 12.5.2 estudiaremos la noción de reducción *polinómica*, que es más grosera que la reducción lineal, pero más fácil de utilizar porque no requiere las nociones que se presentaron en las Definiciones 12.4.2 y 12.4.3. Además, es más general.

12.4.2 Reducciones entre problemas matriciales

Recuerde que una matriz $m \times n$ es *cuadrada* si $m = n$. Una *matriz triangular superior* es una matriz cuadrada M cuyos coeficientes por debajo de la diagonal son todos nulos: $M_{ij} = 0$ cuando $i > j$. La *traspuesta* de una matriz $m \times n$ es la matriz $n \times m$ M^t, definida por $M_{ij} = M_{ji}$. Una matriz M es *simétrica* si es igual a su propia traspuesta, lo cual implica que tiene que ser cuadrada. Una matriz cuadrada M es *no singular* si existe una matriz N tal que $M \times N = I$, la matriz identidad; esta matriz N se denomina la *inversa* de M y se denota en la forma M^{-1}.

Vimos en la Sección 7.6 que basta un tiempo en $O(2^{2,81})$ —o incluso en $O(2^{2,376})$— para multiplicar dos matrices arbitrarias $n \times n$, en contra de la intuición que sugiere que este problema requiere inevitablemente un tiempo que está en $\Omega(n^3)$. ¿Es posible que la multiplicación de matrices triangulares superiores se pudiera efectuar

de forma significativamente más rápida que la multiplicación de matrices cuadradas arbitrarias? ¿Qué sucede con las matrices simétricas? Hay algo más interesante, y es que la experiencia podría hacernos creer que la inversión de matrices no singulares es inherentemente más difícil que su multiplicación.

Denotaremos los problemas de multiplicación de matrices cuadradas arbitrarias, multiplicación de matrices triangulares superiores, multiplicación de matrices simétricas e inversión de matrices no singulares triangulares superiores mediante las siglas **MQ**, **MT**, **MS** e **IT** respectivamente. Demostraremos, haciendo suposiciones razonables, que estos cuatro problemas son linealmente equivalentes. El problema de invertir una matriz no singular *arbitraria* también es linealmente equivalente a estos problemas, pero la demostración es más difícil y requiere una suposición más fuerte; véase el problema 12.31. Obsérvese, sin embargo que el algoritmo resultante es numéricamente inestable. consiguientemente, podemos invertir cualquier matriz no singular $n \times n$ en un tiempo que está en $O(n^{2,376})$ —al menos en teoría. Además, cualquier algoritmo nuevo para multiplicar matrices triangulares o simétricas más eficientemente proporcionará también un algoritmo nuevo, más eficiente, para invertir matrices arbitrarias no singulares.

En lo que sigue, mediremos la complejidad de los algoritmos que manipulan matrices $n \times n$ en términos de n, denominando cuadrático, por ejemplo, a un algoritmo que funcione en un tiempo que esté en $\Theta(n^2)$. Hablando formalmente, esto es incorrecto porque el tiempo de ejecución debería darse en función del *tamaño* del caso, así que un tiempo que esté en $\Theta(n^2)$ es realmente lineal. Todos los problemas que consideremos aquí son al menos cuadráticos en el caso peor, porque cualquier algoritmo que los resuelva debe de examinar todos los coeficientes de la matriz o matrices implicadas. Pasamos ahora a demostrar la equivalencia lineal de nuestros problemas matriciales mediante una sucesión de reducciones: demostramos que $MT \leq' MQ$, $MS \leq' MQ$, $MQ \leq' MT$, $MQ \leq' MS$, $MQ \leq' IT$, y $IT \leq' MQ$.

Esto implica que $MQ =' MT$, $MQ \equiv' MS$ y $MQ =' IT$. La equivalencia de los cuatro problemas se sigue inmediatamente de la transitividad de las equivalencias lineales.

Teorema 12.4.5. $MT \leq' MQ$ y $MQ \leq' MS$.

Demostración: Todo algoritmo que pueda multiplicar dos matrices cuadradas arbitrarias se puede utilizar directamente para multiplicar matrices triangulares superiores y matrices simétricas.

Teorema 12.4.6. $MQ \leq' MT$, *suponiendo que* MT *sea suave.*

Demostración: Supongamos que existe un algoritmo capaz de multiplicar dos matrices triangulares superiores $n \times n$ en un tiempo $O(t(m) \mid m$ es una potencia de 2), donde $t(m)$ es una función suave. Sean A y B dos matrices arbitrarias $n \times n$ que haya que multiplicar. Considere el siguiente producto de matrices triangulares superiores $3n \times 3n$:

$$\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & AB \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

En donde "0" denota la matriz $n \times n$ cuyos coeficientes son todos ceros. Este producto nos muestra la forma de obtener el resultado deseado, AB, mediante una reducción a una multiplicación más grande de matrices triangulares superiores. El tiempo necesario para esta operación está en $O(n^2)$ para la preparación de las dos matrices triangulares superiores y la extracción de AB de su producto, más $O(t(3n))$ para la multiplicación de las dos matrices triangulares superiores. Por la hipótesis de suavidad, $t(3n) \in O(t(n))$. Dado que $t(n)$ es al menos cuadrática, $t(n) \in \Omega(n^2)$ y por tanto $n^2 \in O(t(n))$. Consiguientemente, el tiempo total requerido para multiplicar matrices arbitrarias $n \times n$ está en $O(n^2 + t(3n))$, que es lo mismo que $O(t(n))$.

Teorema 12.4.7 *$MQ \leq^t MS$, suponiendo que* MS *sea suave*

Demostración: Es parecida a la demostración del Teorema 12.4.6: reducimos la multiplicación de dos matrices arbitrarias $n \times n$ A y B a la multiplicación de matrices simétricas $2n \times 2n$:

$$\begin{pmatrix} 0 & A \\ A' & 0 \end{pmatrix} \times \begin{pmatrix} 0 & B' \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} AB & 0 \\ 0 & A'B' \end{pmatrix}$$

Dejaremos los detalles para el lector. Obsérvese que el producto de dos matrices simétricas no es necesariamente simétrico.

Teorema 12.4.8 *$MQ \leq^t IT$, suponiendo que* IT *sea suave*

Demostración: Supongamos que existe un algoritmo capaz de invertir una matriz triangular superior $n \times n$ no singular en un tiempo que está en $O(t(n))$, donde $t(n)$ es una función suave. Sean A y B dos matrices arbitrarias $n \times n$ que haya que multiplicar. Considere el siguiente producto de matrices triangulares $3n \times 3n$:

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix} \times \begin{pmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix}$$

donde I es la matriz identidad $n \times n$. El producto nos muestra la forma de obtener el producto deseado AB invirtiendo la primera de las matrices triangulares superiores anteriores: el resultado aparece en la esquina superior derecha de la inversa.

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^{-1} = \begin{pmatrix} I & -A & AB \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

Tal como en la demostración del Teorema 12.4.6, toda esta operación requiere un tiempo que está en $O(n^2 + t(3n))$, que es lo mismo que $O(t(n))$.

Lema 12.4.9 *$IT \leq^t IT2$, suponiendo que* IT *sea suave.*

Queda por demostrar que $IT \leq^t MQ$, que es la más interesante de todas estas reducciones. Para ello es útil introducir otro problema más: IT2 es el problema consistente en invertir matrices no singulares triangulares superiores cuyo tamaño sea una potencia de dos.

Demostración: Supongamos que existe un algoritmo capaz de invertir una matriz no singular triangular superior $m \times m$ en un tiempo que esté en $O(t(m) \mid m$ es una potencia de 2), donde $t(m)$ es una función suave. Sea A una matriz triangular superior $n \times n$ con n arbitrario. Sea m la menor potencia de 2 que no sea menor que n. Sea B la matriz triangular superior $m \times m$ tal que $B_{ij} = A_{ij}$ para $1 \leq i \leq n$ y $1 \leq j \leq n$, $B_{ii} = 1$ para $n < i \leq m$ y $B_{ij} = 0$ en caso contrario:

$$B = \begin{pmatrix} A & 0 \\ 0 & I \end{pmatrix}$$

donde los ceros son matrices nulas rectangulares del tamaño adecuado para que B sea $m \times m$, y I es la matriz identidad $(m-n) \times (m-n)$. Es fácil verificar que la inversa de A se puede leer directamente como la submatriz $n \times n$ que aparece en la esquina superior izquierda de B^{-1}. Por tanto, el cálculo de A^{-1} requiere un tiempo que está en $O(t(m))$ para invertir B, más otro tiempo en $O(n^2)$ para preparar la matriz B y leer la respuesta de B^{-1}. Dado que $t(m)$ es una función suave, y m es par, $t(m) = t(2(m/2)) \leq ct(m/2)$ para alguna constante adecuada c, siempre y cuando n sea suficientemente grande. Dado que las funciones suaves son asintóticamente no decrecientes, y $m/2 < n$, $t(m/2) \leq t(n)$, una vez más siempre y cuando n sea suficientemente grande. Se sigue que $t(m) \leq ct(n)$. Por tanto la matriz A se puede invertir en un tiempo que está en $O(ct(n) + n^2)$. Esto es lo mismo que $O(t(n))$ porque $t(n)$ es al menos cuadrática.

Lema 12.4.10 *$IT2 \leq^t MQ$, suponiendo que* MQ *sea fuertemente cuadrático.*

Demostración: Sea A una matriz no singular triangular superior $n \times n$ que haya que invertir, donde n es una potencia de 2. Si n=1, la inversión es trivial. En caso contrario, se descompone A en tres submatrices B, C y D, cada una de las cuales es de tamaño $\frac{n}{2} \times \frac{n}{2}$, definidas por

$$A = \begin{pmatrix} B & C \\ 0 & D \end{pmatrix}.$$

Obsérvese que B y D son triangulares superiores, mientras que C es arbitraria. De manera similar, sean F, G y H matrices desconocidas $\frac{n}{2} \times \frac{n}{2}$ tales que

$$A^{-1} = \begin{pmatrix} F & G \\ 0 & H \end{pmatrix}.$$

La submatriz inferior izquierda es cero porque la inversa de una matriz superior no singular triangular también es triangular superior; véase el problema 12.30. El producto de A por A^{-1} debería de ser la matriz identidad.

$$\begin{pmatrix} B & C \\ 0 & D \end{pmatrix} \times \begin{pmatrix} F & G \\ 0 & H \end{pmatrix} = \begin{pmatrix} BF & BG+CH \\ 0 & DH \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix}$$

Por tanto, $BF = DH = I$, lo cual implica que B y D son no singulares y que $F=B^{-1}$ y $O=B^{-1}$. Además, $BG + CH = 0$ y por tanto $G = -B^{-1}CH = -B^{-1}CD^{-1}$. Reuniendo todo esto, la inversa de A se obtiene a partir de las inversas de B y D al cabo de dos multiplicaciones matriciales para calcular $-B^{-1}CD^{-1}$:

$$A^{-1} = \begin{pmatrix} B^{-1} & -B^{-1}CD^{-1} \\ 0 & D^{-1} \end{pmatrix}.$$

Dado que B y D son matrices no singulares triangulares superiores y de tamaño igual a la mitad del tamaño de A, esto sugiere un algoritmo del tipo divide y vencerás para calcular A^{-1} mediante dos cálculos recursivos de inversas, dos multiplicaciones matriciales y unas cuantas operaciones de mantenimiento que requieren un tiempo despreciable $g(n) \in O(n^2)$.

Sea $I(n)$ el tiempo invertido por este algoritmo para calcular la inversa de una matriz triangular superior $n \times n$ cuando n es una potencia de 2. Sea $LM(n)$ el tiempo necesario para multiplicar dos matrices arbitrarias $n \times n$. A partir de lo dicho anteriormente, $I(n) \leq 2I(n/2) + 2M(n/2) + g(n)$ cuando n es una potencia de 2 mayor que 1. Por la suposición de que **MQ** es fuertemente cuadrático, $M(n) \in \Theta(t(n))$ para alguna función fuertemente cuadrática $t(n)$. Sean a, b y c constantes tales que $g(n) \leq an^2$, $t(n) \geq bn^2$ y $M(n) \leq ct(n)$ para todo n suficientemente grande. Existe la constante b porque todas las funciones fuertemente cuadráticas son al menos cuadráticas por el problema 12.23. Por definición de fuertemente cuadrática, $t(n) \geq 4t(n/2)$ y por tanto $t(n/2) \leq \frac{1}{4}t(n)$. De todas estas fórmulas se sigue que

$$\begin{aligned} I(n) &\leq 2I(n/2) + 2M(n/2) + g(n) \\ &\leq 2I(n/2) + 2ct(n/2) + an^2 \\ &\leq 2I(n/2) + \left(\tfrac{c}{2} + \tfrac{a}{b}\right)t(n) \\ &\leq 2I(n/2) + dt(n) \end{aligned} \tag{12.2}$$

para todo $n \geq n_0$ que sea una potencia de 2, y para unas constantes adecuadas d y n_0. Sin pérdida de generalidad, podemos hacer que n_0 sea una potencia de 2.

Queda por demostrar que $I(n) \in O(t(n) \mid n$ es una potencia de 2). Para esto emplearemos inducción constructiva, con objeto de hallar una constante u tal que $I(n) \leq ut(n)$ para todos los $n \geq n_0$ que sean potencias de 2. La base de la inducción está demostrada, siempre y cuando seleccionemos $u \geq I(n_0)/t(n_0)$. Para el paso de inducción, se considera cualquier que sea una potencia de dos, y supongamos la hipótesis de inducción parcialmente especificada de que $I(n/2) \leq ut(n/2)$. Por la Ecuación 12.2, la hipótesis de inducción y el hecho de que $t(n/2) \leq \frac{1}{4}t(n)$,

$$\begin{aligned} I(n) &\le 2I(n/2) + dt(n) \\ &\le 2ut(n/2) + dt(n) \\ &\le \left(\tfrac{u}{2} + d\right)t(n). \end{aligned}$$

Esto muestra que $I(n) \le ut(n)$ siempre y cuando $\frac{u}{2} d \le u$, que es lo mismo que $u \ge 2d$. En conclusión, $I(n) \le ut(n)$ es válido para todos los $n \ge n_0$ que sean potencias de 2, siempre y cuando seleccionemos $u \ge \max(I(n_0) / t(n_0), 2d)$. Esto completa la demostración de que $I(n) \in O(t(n) \mid n$ es una potencia de 2), y por tanto que $IT2 \le^t MQ$ suponiendo que **MQ** sea fuertemente cuadrático.

La reducción utilizada en esta demostración es distinta de las reducciones vistas anteriormente, en el sentido de que una única inversión de una matriz triangular superior implica un elevado número de multiplicaciones matriciales, si se cuentan las multiplicaciones hechas por las llamadas recursivas. La linealidad de la reducción solo es posible porque la mayoría de las multiplicaciones implicadas se efectúan en matrices mucho más pequeñas que la que intentamos invertir.

Teorema 12.4.11 *$IT \le^t MQ$, suponiendo que sea* **MQ** *fuertemente cuadrático.*

Demostración: Es casi inmediata a partir de los dos lemas precedentes. El único problema técnico es que necesitamos la suposición de que IT2 es suave para aplicar el Lema 12.4.9, y esto no es consecuencia del Lema 12.4.10 aun cuando **MQ** sea fuertemente cuadrático. Sin embargo, todo va bien, porque la demostración del Lema 12.4.10 se conforma con multiplicar matrices de tamaño $\frac{n}{2} \times \frac{n}{2}$ para invertir una matriz triangular superior de tamaño $n \times n$ cuando n es una potencia de 2. Entonces se puede refinar la Ecuación 12.2 de la forma siguiente:

$$I(n) \le 2I(n/2) + \hat{d}t(n/2)$$

y de aquí se sigue que existe una constante $\hat{u}$ tal que $I(n) \le \hat{u}t(n/2)$ supuesto que n sea una potencia de 2 suficientemente grande. La demostración del Lema 12.4.9 es válida entonces sin necesidad de que $t(n)$ sea suave: es suficiente que $t(n)$ sea asintóticamente no decreciente, lo cual es cierto por ser fuertemente cuadrática. Dejamos los detalles para el lector.

12.4.3 Reducciones entre problemas de caminos mínimos

En esta sección, $\mathbb{R}^\infty$ denota $\mathbb{R}^{\ge 0} \cup \{\infty\}$, con las convenciones naturales de que $x + \infty = \infty$ y $\min(x, \infty) = x$ para todo $x \in \mathbb{R}^\infty$.

Sean X, Y y Z tres conjuntos de nodos. Sea $f: X \times Y \to \mathbb{R}^\infty$ y $g: Y \times Z \to \mathbb{R}^\infty$ dos funciones que representan el coste de ir directamente de un nodo a otro. Un coste infinito representa la ausencia de un enlace directo. Denotaremos mediante $f \times g$ la función $h: X \times Z \to \mathbb{R}^\infty$ definida para todo $x \in X$ y para todo $y \in Y$ por

$$h(x,z) = \min_{y \in Y}\left(f(x,y) + g(y,z)\right)$$

Este es el coste mínimo de ir desde x hasta z pasando exactamente por un nodo de Y. Obsérvese la analogía entre esta definición y la multiplicación matricial ordinaria: la adición y la multiplicación son sustituidas por la operación mínimo y la adición, respectivamente.

La notación precedente resulta especialmente interesante cuando los conjuntos X, Y y Z, y también las funciones f y g, coinciden. En este caso $f \times f$, que denotaremos como f^2, nos da el coste mínimo de ir de un nodo de X a otro también de X (incluso posiblemente el mismo nodo) pasando exactamente por un nodo intermedio (que también puede ser él mismo). De forma similar, $\min(f, f^2)$ da el coste mínimo de ir de un nodo de X hasta otro bien directamente o bien pasando exactamente por un nodo intermedio. El significado de f^i es similar para todo $i > 0$. Es natural definir f^0 como el coste de ir de un nodo a otro permaneciendo en el mismo lugar.

$$f^0(x,y) = \begin{cases} 0 & \text{si } x = y \\ \infty & \text{en caso contrario} \end{cases}$$

El coste mínimo de ir de un nodo a otro sin restricciones sobre el número de nodos que haya en el camino, y que denotaremos f^*, es por tanto

$$f^*(x,y) = \min_{i \geq 0} f^i(x,y)$$

Esta definición implica aparentemente un cálculo infinito; ni siquiera está claro de forma inmediata que f^* esté bien definida. Sin embargo, f nunca toma valores negativos. Todo camino que pase dos veces por el mismo nodo se puede acortar, por tanto, eliminando el bucle que se forma de esta manera, sin incrementar el coste del camino resultante. Consiguientemente, basta considerar solamente aquellas rutas cuya longitud sea menor que el número de nodos que hay en X. De esta manera, se tiene que

$$f^*(x,y) = \min_{0 \leq i \leq n} f^i(x,y) \tag{12.3}$$

El algoritmo evidente para calcular $f \times g$ requiere un tiempo en $\Theta(n^3)$ si los tres conjuntos de nodos tienen el cardinal n y si contamos las sumas y las comparaciones con un coste unitario. Desafortunadamente, no existe una manera evidente de adaptar a este problema el algoritmo de Strassen para la multiplicación de matrices ordinaria (véase la Sección 7.6). La dificultad estriba en que el algoritmo de Strassen hace sustracciones, que son la inversa de las adiciones; no hay un equivalente de esta operación en el presente contexto, puesto que las sumas ordinarias se sustituyen por tomar el mínimo, lo cual no es una operación reversible. Se conocen algoritmos que son asintóticamente más rápidos que $\Theta(n^3)$ para este problema, pero son bastante complicados y solamente tienen ventajas teóricas; véase la Sección 12.8.

La Ecuación 12.3 da lugar a un algoritmo directo para calcular f^* en n veces el tiempo que se necesita para un solo cálculo del tipo $f \times g$. Se trata de un tiempo en

$\Theta(n^4)$ si se utiliza el algoritmo normal. De esta forma, calcular f^* para una función dada f parece, a primera vista, requerir más tiempo que calcular el simple producto $f \times g$. Sin embargo, vimos en la Sección 8.5 un algoritmo de programación dinámica para calcular caminos mínimos dentro de un grafo, a saber, el algoritmo de Floyd; que no es otra cosa que el cálculo de f^*. Por tanto, *es* posible después de todo lograr un tiempo que esté en $\Theta(n^3)$. ¿Podría ser que los problemas de calcular $f \times g$ y f^* fueran de igual complejidad? Sorprendentemente, la respuesta es sí: estos dos problemas son linealmente equivalentes. La existencia de algoritmos asintóticamente más eficientes que $\Theta(n^3)$ para resolver el problema de calcular $f \times g$ implica por tanto que el algoritmo de Floyd para calcular el camino mínimo no es óptimo, al menos en teoría.

Aquí nos conformaremos con enunciar formalmente el teorema principal, y relegamos su demostración a los ejercicios. Suponiendo que f y g estén definidas en el mismo dominio, denotemos por **MUL** y **TRC** los problemas consistentes en calcular $f \times g$ y f^*, respectivamente (**TRC** quiere decir "cierre transitivo reflexivo"). Por sencillez, se medirán las complejidades temporales como función del número de nodos; un algoritmo tal como el de Floyd, por ejemplo, se considera cúbico porque requiere un tiempo que está en $\Theta(n^3)$, aun cuando esto sea incorrecto desde un punto de vista formal porque el tamaño de un caso de n nodos está en $\Theta(n^2)$ si se proporciona en forma de matriz de distancias.

Teorema 12.4.12. **MUL** $\equiv'$ **TRC**, *suponiendo que ambos problemas sean suaves y suponiendo que* **MUL** *sea fuertemente cuadrático.*

Demostración: Hacer los problemas 12.32 y 12.33.

Cuando el intervalo de funciones de coste se restringe a $\{0, \infty\}$, el cálculo de f^* se reduce a determinar para cada par de nodos, si existe o no un camino que los une. Vimos en el problema 8.18 que el algoritmo de Warshall resuelve este problema en un tiempo que está en $\Theta(n^3)$. Sean **MULB** y **TRCB** los problemas consistentes en calcular $f \times g$ y f^*, respectivamente, cuando las funciones de coste se restringen de esta manera. Está claro que **MULB** $\leq'$ **MUL** y que **TRCB** $\leq'$ **TRC**, puesto que los algoritmos generales también pueden utilizarse para resolver casos de los problemas restringidos. Además, la demostración de que **MUL** $\equiv'$ **TRC** se puede adaptar fácilmente para demostrar que **MULB** $\equiv'$ **TRCB** con unas suposiciones similares. Este resultado es interesante como consecuencia del teorema siguiente, que lo relaciona con el problema **MQ** de multiplicación de matrices cuadradas ordinarias, y que estudiamos en la Sección 12.4.2.

Teorema 12.4.13 **MULB** $\leq'$ **MQ**

Demostración: Sea $f \times g$ un caso de tamaño n del problema **MULB**. Supongamos, sin pérdida de generalidad, que los conjuntos de nodos subyacentes son $\{1, 2, ..., n\}$. Definimos dos matrices $n \times n$ llamadas A y B tales que

$$A_{ij} = \begin{cases} 0 & \text{si } f(i,j) = \infty \\ 1 & \text{si } f(i,j) = 0 \end{cases}$$

y análogamente para B con respecto a la función g. Intuitivamente, $A_{ij} = 1$ si y solo si existe un enlace directo f entre los nodos i y j. Por definición de la multiplicación ordinaria de matrices, $(AB)_{ij}$ recuenta el número de formas de alcanzar el nodo j desde el nodo i pasando primero por un enlace f y después por un enlace g. Por tanto,

$$(f \times g)(i,j) = \begin{cases} \infty & \text{si } (AB)_{ij} = 0 \\ 0 & \text{en caso contrario} \end{cases}$$

De esta manera, para resolver un caso de tamaño n del problema **MULB**, lo único que se necesita es una sola multiplicación matricial arbitraria $n \times n$.

El algoritmo de Strassen se puede utilizar, por tanto, para resolver los problemas **MULB** y **TRCB** en un tiempo que está en $O(n^{2,81})$, lo cual demuestra que el algoritmo de Warshall no es óptimo. Sin embargo, la utilización del algoritmo de Strassen requiere un número de operaciones *aritméticas* que está en $O(n^{2,81})$; el tiempo $O(n^3)$ que requiere el algoritmo de Warshall solo cuenta como elementales las operaciones booleanas.

Se produce una situación interesante cuando consideramos versiones *simétricas* de los problemas estudiados en esta sección. Correspondientes a los problemas sobre grafos no dirigidos. Una función de coste f: $X \times X \to \mathbb{R}^\infty$ es simétrica si $f(u, v) = f(v, u)$ para todo $u, v \in X$. Los cuatro problemas descritos anteriormente poseen versiones simétricas que surgen cuando las funciones de coste implicadas son simétricas; los llamaremos **MULS**, **TRCS**, **MULBS** y **TRCBS**. Inspirándose en el Teorema 12.4.7, que demuestra que la multiplicación ordinaria de matrices no es más fácil cuando están implicadas matrices simétricas, resulta tentador conjeturar que lo mismo será cierto con los problemas de caminos mínimos. De hecho, la demostración del Teorema 12.4.7 se puede aplicar *mutatis mutandis* para probar que $MUL \equiv' MULS$ y que $MULB \equiv' MULBS$. Sin embargo, la analogía falla al llegar a **TRCBS**. Si lo piensa un momento, se convencerá de que esto no es más que un nombre ostentoso del problema consistente en hallar las componentes conexas de un grafo no dirigido, problema que se resuelve fácilmente en un tiempo que está en $O(n^2)$ mediante un recorrido en profundidad. Por otra parte, no se conoce ningún algoritmo que pueda resolver **MULB** con tanta rapidez. Pero recuerde que $TRCB \equiv' MULB$. Por tanto, parece que **TRCB** es realmente más difícil que su versión simétrica, **TRCBS**. Por la misma razón, parece que **MULBS** es realmente más difícil que **TRCBS**. Esto es extraño, en cierto sentido, porque el algoritmo directo que se obtiene de la Ecuación 12.3 requeriría resolver n casos de **MULBS** para resolver un solo caso de **TRCBS**.

12.5 INTRODUCCIÓN A LA NP-COMPLETITUD

En la vida real existen muchos problemas prácticos para los cuales no se conoce ningún algoritmo eficiente, pero cuya dificultad intrínseca no ha conseguido demostrar nadie. Entre estos se cuentan problemas tan conocidos como el del viajante de comercio, el coloreado óptimo de grafos, el problema de la mochila, los ciclos hamiltonianos, la programación entera, la búsqueda del camino simple más largo de un grafo, y la satisfacción de una fórmula booleana: algunos de ellos se describen más adelante. ¿Hay que culpar a la algorítmica o a la complejidad? Quizá existan realmente algoritmos eficientes para estos problemas. Después de todo, las ciencias de la computación son relativamente recientes: es claro que quedan por descubrir nuevas técnicas algorítmicas. Por otra parte, quizá estos problemas sean intrínsecamente difíciles, pero carecemos de las técnicas para demostrarlo.

Esta sección presenta un resultado notable: un algoritmo eficiente para resolver cualquiera de los problemas enumerados en el párrafo anterior proporcionaría automáticamente algoritmos eficientes para todos ellos. No sabemos si estos problemas son fáciles o difíciles de resolver, pero ciertamente sabemos que todos ellos son de complejidad parecida. La importancia práctica de estos problemas ha asegurado que cada uno de ellos por separado haya sido objeto de esfuerzos sostenidos para hallar un método de solución eficiente. Por esta razón, la creencia de que no existen tales algoritmos se encuentra ampliamente difundida. Si se tiene que resolver un problema, y uno puede demostrar que es computacionalmente equivalente a uno de los mencionados anteriormente, es posible tomar esto como evidencia convincente—pero no como demostración—de que el problema es difícil en el caso peor. Por lo menos, usted podrá estar seguro de que en este momento nadie sabe resolver eficientemente su problema.

En el núcleo de esta teoría yace la idea de que puede haber problemas que sean realmente difíciles de resolver, pero para los cuales se puede verificar eficientemente la validez de cualquier supuesta solución. Considere el *problema del ciclo hamiltoniano* como ejemplo. Dado un grafo no dirigido $G = <N, A>$, el problema consiste en hallar un camino que comience en algún nodo, visite cada nodo exactamente una vez, y vuelva al nodo de partida. Decimos que un grafo es hamiltoniano si existe uno de estos ciclos. Se cree que el problema es difícil. Sin embargo, es obviamente fácil verificar si una cierta sucesión de nodos define un ciclo hamiltoniano. Otro ejemplo es la factorización. Dado un número compuesto, puede ser difícil hallar un divisor no trivial, pero es sencillo verificar la validez de cualquier supuesto divisor. Quizá parezca evidente que para muchos problemas resulta realmente más sencillo verificar la validez de una supuesta solución que encontrar una solución partiendo de cero. El mayor desconcierto de la moderna teoría de la complejidad computacional es que no sabemos demostrar esto.

12.5.1 Las clases P y NP

Antes de seguir adelante, nos servirá de ayuda definir lo que entendemos por un algoritmo eficiente. ¿Significa esto que requiere un tiempo que está en $O(n \log n)$?

¿En $O(n^2)$? ¿En $O(n^{2,81})$? Todo depende del problema que haya que resolver. Un algoritmo de ordenación que requiere un tiempo en $\Theta(n^2)$ es ineficiente, mientras que un algoritmo de multiplicación matricial que requiera un tiempo en $O(n^2 \log n)$ sería un descubrimiento asombroso. Por tanto, podemos sentir la tentación de decir que un algoritmo es eficiente si es mejor que el algoritmo evidente, o quizá si se trata del mejor algoritmo posible para resolver nuestro problema. Sin embargo, esta definición sería imprecisa y resultaría difícil trabajar con ella, y en algunos casos "el mejor algoritmo posible" ni siquiera existe; véase el problema 12.34. Además, existen problemas para los cuales hasta el mejor algoritmo posible requiere una cantidad de tiempo exorbitante, incluso para casos pequeños. ¿No sería razonable admitir que estos problemas son inherentemente intratables, en lugar de afirmar que unos algoritmos inteligentes son eficientes aunque resulten demasiado lentos para utilizarlos en la práctica?

Para nuestros propósitos actuales, responderemos a esta pregunta estipulando que un algoritmo es *eficiente* si existe un polinomio $p(n)$ tal que el algoritmo puede resolver cualquier caso de tamaño n en un tiempo que está en $O(p(n))$. Diremos que estos algoritmos son de *tiempo polinómico*. Esta definición tiene su origen en la comparación efectuada en la Sección 2.6 entre un algoritmo que requiere un tiempo que está en $\Theta(2^n)$ y uno que solo necesita un tiempo que está en $\Theta(n^3)$, así como en algunos de los ejemplos dados en la Sección 2.7. Los algoritmos de tiempo exponencial se vuelven rápidamente inútiles en la práctica, mientras que en general un algoritmo de tiempo polinómico nos permite resolver casos mucho mayores.

Esta noción de eficiencia no debe de tomarse literalmente. Dados dos algoritmos que requieran un tiempo en $\Theta(n^{\lg\lg n})$ y en $\Theta(n^{10})$ respectivamente, el primero es ineficiente según nuestra definición porque no es de tiempo polinómico. Sin embargo, vencerá al algoritmo de tiempo polinómico en todos los casos de tamaño menor que 10^{300}, suponiendo que las constantes ocultas sean parecidas. De hecho, no es razonable afirmar que un algoritmo que requiera un tiempo en $\Theta(n^{10})$ sea eficiente en la práctica. Sin embargo, decretar que es eficiente mientras que $\Theta(n^4)$ no lo es, por ejemplo, sería excesivamente arbitrario. Además, incluso un algoritmo de tiempo lineal puede no ser utilizable en la práctica si la constante multiplicativa oculta es demasiado grande, mientras que un algoritmo que requiera un tiempo exponencial en el caso peor puede resultar sumamente rápido en la mayoría de los casos. Sin embargo, existen ventajas técnicas significativas cuando se considera la clase de todos los algoritmos de tiempo polinómico. En particular, todos los modelos razonables de computación determinista con un solo procesador se pueden simular entre sí con una pérdida de velocidad que es, como mucho, polinómica. Por tanto, la noción de computabilidad en tiempo polinómico es robusta: no depende del modelo que se prefiera, a no ser que se utilicen modelos posiblemente más potentes como las computadoras probabilistas o cuánticas. Además, el hecho de que las sumas , productos y composiciones de polinomios sean a su vez polinomios resultará útil.

En esta sección, todos nuestros análisis del tiempo requerido por un algoritmo van a ser "salvo un polinomio". Esto significa que no dudaremos en contar como

de coste unitario una operación que requiera realmente una cantidad de tiempo polinómica. Por ejemplo, podemos contar las sumas y multiplicaciones con un coste unitario, aun en aquellos operandos cuyo tamaño crece con el tamaño del caso que se está considerando, siempre y cuando este crecimiento esté acotado por algún polinomio. Esto es admisible porque solo deseamos distinguir los algoritmos de tiempo polinómico frente a los que no son de tiempo polinómico, y porque se necesita un tiempo polinómico para ejecutar un número polinómico de operaciones de tiempo polinómico; véase el problema 12.35. Por otra parte, no contaremos con un coste unitario la aritmética que implique operandos de tamaño exponencialmente mayor que el del caso. Si el algoritmo necesita operandos tan grandes, es preciso descomponerlos en segmentos, guardarlos en un vector, e invertir el tiempo necesario para efectuar la aritmética de precisión múltiple; tales algoritmos no pueden ser de tiempo polinómico.

Nuestro objetivo es distinguir aquellos problemas que se pueden resolver eficientemente de aquellos en que esto no es posible. Por razones técnicas nos concentramos en el estudio de problemas de *decisión*.

Para estos, la respuesta es *si* o *no*, o lo que es equivalente, *verdadero* o *falso*. Por ejemplo, "buscar un ciclo hamiltoniano en G" no es un problema de decisión, pero "¿Es hamiltoniano el grafo G"? si que lo es. Se puede pensar que un problema de decisión define un conjunto X de casos en los que la respuesta correcta es "si". A estos los llamamos *casos-si*, todos los demás casos son *casos-no*. Diremos que un algoritmo correcto que resuelva el problema de decisión *acepta* el caso si y *rechaza* los casos-no.

Definición 12.5.1 P *es la clase de problemas de decisión que se pueden resolver mediante un algoritmo de tiempo polinómico.*

Por sencillez, no admitimos algoritmos probabilistas en esta decisión, aun cuando los del tipo Las Vegas cuyas respuestas tienen la corrección garantizada. Esto es una razón más para no tomar literalmente la definición: pueden existir problemas de decisión que se puedan resolver en un tiempo esperado polinómico, pero solo mediante algoritmos probabilistas; tales problemas no se encuentran en P según la definición.

La teoría de la NP-completitud estudia la noción de propiedades *verificables* polinómicamente. Intuitivamente, un problema de decisión X es verificable eficientemente si un ser omnisciente pudiera producir una evidencia convincente de que $x \in X$ siempre que así fuera. Dada esta evidencia, deberíamos ser capaces de verificar eficientemente que en verdad $x \in X$ sin más contacto con el mencionado ser. Sin embargo, si $x \in X$, no deberíamos quedar falsamente convencidos de que $x \in X$ independientemente de lo que nos diga el ser. Considere como ejemplo el problema consistente en decidir si un grafo es hamiltoniano. Aun cuando se cree que este problema es difícil, es verificable eficientemente: si el ser nos muestra un ciclo hamiltoniano, es sencillo verificar si es o no correcto. Por otra parte, nada de lo

que nos muestre el ser —con la posible excepción de una escopeta— podrá convencernos de que el grafo es hamiltoniano si no lo es realmente.

Considere un problema de decisión X. Sea Q un conjunto, arbitrario por el momento, al que llamaremos *espacio de prueba* para X. Un *sistema de prueba* para X es un conjunto de F parejas $<x, q>$. Para todo $x \in X$, tiene que existir al menos un $q \in Q$ tal que $<x, q> \in F$; por otra parte, cuando $x \notin X$, no debe de existir ninguno de estos q. Por tanto, basta que el ser nos muestre algún $q \in Q$ tal que $<x, q> \in F$ para convencernos de que $x \in X$. Al ver este q quedaremos convencidos de que x es un caso-si, dado que si x fuera un caso-no, no existiría ningún q como este. Además, el ser siempre puede mostrar uno de estos q para los casos-si porque siempre existen por definición. Formalmente, F es un subconjunto de $X \times Q$ tal que

$$(\forall x \in X)(\exists q \in Q)[\langle x, q\rangle \in F].$$

Todo q tal que $<x, q> \in F$ se denomina *prueba* o bien *certificado* de que $x \in X$. No hemos especificado explícitamente en la definición formal anterior que

$$(\forall x \notin X)(\forall q \in Q)[\langle x, q\rangle \notin F]$$

porque está implícito en el requisito de que F tiene que ser un subconjunto de $X \times Q$.

Por ejemplo, si X es el conjunto de todos los grafos hamiltonianos, podemos tomar como Q el conjunto de sucesiones de nodos del grafo, y definimos $<G, s> \in F$ si y solo si la sucesión de nodos s especifica un ciclo hamiltoniano en el grafo G.

Otro ejemplo más: si X es el conjunto de todos los números compuestos, podemos tomar $Q = \mathbb{N}$ como espacio de pruebas y

$$F=\{\langle n, q\rangle \mid 1 < q < n \quad \text{y} \quad q \text{ divide a } n\}$$

como sistema el de pruebas. Este sistema de prueba no es único. Otra posibilidad podría ser

$$F' = \{\langle n, q\rangle \mid 1 < q < n \quad \text{y} \quad \text{mcd}(q,n) \neq 1\}$$

Se pueden obtener más sistemas de pruebas para el mismo problema a partir de la discusión de la Sección 10.6.2, que muestra que los certificados de que un número es compuesto pueden no servir de ayuda para factorizarlo.

La clase NP corresponde a los problemas de decisión que tienen un sistema de prueba *eficiente*, lo cual significa que todo caso-si debe poseer al menos un certificado *sucinto*, cuya validez pueda ser verificada rápidamente.

Definición 12.5.2 *NP es la clase de problemas de decisión que admiten un sistema de pruebas $F \subseteq X \times Q$ tal que existe un polinomio $p(n)$ y un algoritmo de tiempo polinómico A tal que*

- ◊ *Para todo $x \in X$ existe un $q \in Q$ tal que $\langle x, q\rangle \in F$ y además el tamaño de q es como máximo $p(n)$, donde n es el tamaño de x.*
- ◊ *Para todos los pares $\langle x, q\rangle$, el algoritmo A puede verificar si $\langle x, q\rangle$ pertenece o no a F. En otras palabras, $F \in P$.*

Los dos ejemplos anteriores encajan en esta definición. Un ciclo hamiltoniano en un grafo, si existe, requiere menos espacio para describirlo que el grafo, y se puede verificar en un tiempo lineal si una sucesión de nodos forma un ciclo hamiltoniano. De forma similar, todo factor no trivial de un número compuesto es menor que ese número, y basta una sola división para verificar su validez. Por tanto, estos dos problemas están en NP.

Antes de seguir adelante, hay que hacer una advertencia seria: quizá se sienta tentado a pensar que las letras NP quieren decir "no polinómico". ¡No es así! Esto sería una tontería porque (1) todo problema que se pueda resolver en un tiempo polinómico está automáticamente en NP (Teorema 12.5.3) y (2) no conocemos la forma de demostrar la existencia de algún problema en NP que no se pueda resolver en tiempo polinómico, aunque conjeturamos que existen. De hecho, NP quiere decir "tiempo polinómico no determinista", tal como se verá en la Sección 12.5.6.

Otro posible asunto problemático es que la definición de NP es asimétrica: requerimos la existencia de certificados sucintos para todos los casos-si, pero no existe ese requisito para los casos-no. Aun cuando el conjunto de todos los grafos hamiltonianos está claramente en NP, no existe tal evidencia para el conjunto de todos los grafos *no* hamiltonianos. De hecho, ¿qué clase de evidencia sucinta podría convencernos eficientemente de que un grafo *no* es hamiltoniano cuando lo es? Se conjetura que en general no puede existir tal evidencia, y por tanto que el conjunto de todos los grafos no hamiltonianos no está en NP. El problema 12.36 ofrece una sorpresa de este estilo.

Nuestro primer teorema establece una relación entre P y NP

Teorema 12.5.3 $P \subseteq NP$.

Demostración: Intuitivamente, este resultado se debe a que no necesitamos ayuda de un ser omnisciente cuando podemos manejar por nosotros mismos el problema de decisión. Formalmente, considérese un problema de decisión arbitrario $X \in$ P. Sea $Q = \{0\}$ un espacio de pruebas trivial. Definimos

$$F = \{\langle x, 0\rangle \mid x \in X\}.$$

Claramente, todo caso-si admite un "certificado" sucinto, a saber, 0, y los casos-no carecen por completo de certificados. Además, basta verificar que $x \in X$ para establecer que $\langle x, q\rangle \in F$. Esto se puede hacer en tiempo polinómico precisamente porque hemos supuesto que $X \in$ P.

La cuestión fundamental que sigue abierta es si la inclusión de conjuntos del Teorema 12.5.3 es o no propia. ¿Es posible que P = NP? Si tal cosa ocurriese, toda propiedad que pudiera verificarse en tiempo polinómico dado un certificado, podría decidirse también en tiempo polinómico partiendo de cero. Aunque esto parece improbable, nadie ha sido capaz de zanjar esta cuestión. En el resto de esta sección analizaremos las consecuencias de la *conjetura*

$$\mathrm{P} \neq \mathrm{NP}.$$

Para esto, necesitamos una noción de reducción que nos permita comparar la dificultad intrínseca de los problemas NP y descubrir que hay problemas en NP que son tan difíciles como cualquier otro problema de NP. Tales problemas, que se denominan *NP-completos*, se pueden resolver en tiempo polinómico si y solo si todos los demás problemas de NP se pueden resolver también en tiempo polinómico, lo cual equivale a decir que P = NP. Por tanto, con la conjetura P ≠ NP, sabemos que los problemas NP-completos no se pueden resolver en tiempo polinómico.

12.5.2 Reducciones polinómicas

La noción de reducción lineal y de equivalencia lineal que se consideraban en la Sección 14.4 son interesantes para problemas que se pueden resolver en tiempo cuadrático o cúbico. Sin embargo, es demasiado restrictiva cuando consideramos problemas para los cuales los mejores algoritmos conocidos requieren un tiempo exponencial. Por esta razón, presentamos una clase distinta de reducción.

Definición 12.5.4 *Sean* **A** *y* **B** *dos problemas. Diremos que* **A** es polinómicamente Turing reducible a **B** *si existe un algoritmo para resolver* **A** *en un tiempo que sería polinómico si pudiéramos resolver casos arbitrarios del problema* **B** *con coste unitario. Esto se denota* $\mathbf{A} \leq_T^p \mathbf{B}$. *Cuando* $\mathbf{A} \leq_T^p \mathbf{B}$ *y* $\mathbf{B} \leq_T^p \mathbf{A}$ *son ciertos ambos, decimos que* **A** y **B** son polinómicamente Turing equivalentes *y escribimos* $\mathbf{B} \equiv_T^p \mathbf{A}$.

En otras palabras, el algoritmo para resolver el problema **A** puede utilizar como le plazca un algoritmo imaginario que puede resolver el problema **B** con coste unitario. Este algoritmo imaginario se denomina a veces *oráculo*. Como en el caso lineal, la demostración por reducción suele adoptar la forma de un algoritmo explícito para resolver un problema mediante llamadas a un algoritmo arbitrario para el otro problema. Una vez más, esto se podría utilizar para enseñar a alguien que sepa resolver uno de los problemas a resolver el otro. Y también de nuevo, las reducciones polinómicas son transitivas: si $\mathbf{A} \leq_T^p \mathbf{B}$ y $\mathbf{B} \leq_T^p \mathbf{C}$, entonces $\mathbf{A} \leq_T^p \mathbf{C}$. A diferencia de las reducciones lineales, sin embargo, permitimos que el primer algoritmo requiera una cantidad polinómica de tiempo, contando las llamadas al segundo algoritmo como de coste unitario, y llamar al segundo algoritmo un número polinómico de veces para casos arbitrarios de tamaño polinómico respecto al tamaño del ejemplar original.

Como primer ejemplo, demostraremos la equivalencia polinómica de las dos versiones del problema del ciclo hamiltoniano. Denotemos mediante **HAM** y **HAMD** los problemas de hallar un ciclo hamiltoniano en un grafo si este existe y de decidir si un grafo es o no hamiltoniano, respectivamente. Permitiremos que un algoritmo para **HAM** devuelva una respuesta arbitraria cuando se le presente un grafo no hamiltoniano. El teorema siguiente dice que no es significativamente más difícil hallar un ciclo hamiltoniano que decidir si un grafo es o no hamiltoniano.

Teorema 15.5.5 $\mathbf{HAM} \equiv_T^p \mathbf{HAMD}$

Demostración: Primero demostramos el sentido evidente: **HAMD** $\leq_T^P$ **HAM**. Considere el algoritmo siguiente:

```
función HamD(G : grafo)
    σ ← Ham(G)
    si σ define un ciclo hamiltoniano en G
        entonces devolver verdadero
        sino devolver falso
```

Este algoritmo resuelve **HAMD** correctamente, siempre y cuando el algoritmo *Ham* resuelva el problema **HAM** correctamente: por definición de **HAM**, el algoritmo *Ham* debe devolver un ciclo hamiltoniano de G si lo hubiere, en cuyo caso *HamD* devolverá, correctamente, el valor *verdadero*. A la inversa, si el grafo no es hamiltoniano, la salida s que devuelve *Ham* no puede ser un ciclo hamiltoniano, y por tanto *HamD* devolverá, correctamente, el valor *falso*. Está claro que *HamD* requiere un tiempo polinómico siempre y cuando contemos las llamadas a *Ham* con coste unitario.
Consideremos ahora la dirección interesante: **HAM** $\leq_T^P$ **HAMD.** Tenemos que *buscar* un ciclo hamiltoniano suponiendo que sepamos *decidir* si existen tales ciclos. La idea es considerar por turnos todas las aristas. Para cada una, preguntamos si el grafo seguiría siendo hamiltoniano si se eliminase esa arista. Solo mantenemos la arista si su eliminación haría que el grafo fuera no hamiltoniano; en caso contrario la eliminamos y pasamos a la arista siguiente. El grafo resultante seguirá siendo hamiltoniano, puesto que nunca hacemos un cambio que pudiera destruir esta propiedad. Además, contiene solamente las aristas necesarias para definir un ciclo hamiltoniano, porque cualquier arista adicional se podría eliminar sin hacer que el grafo fuera no hamiltoniano, y por tanto habría sido eliminado al llegar su turno. Por tanto, basta seguir las aristas del grafo final para obtener un ciclo hamiltoniano en el grafo original. Véase un esbozo de este algoritmo voraz.

```
función Ham(G = ⟨N, A⟩)
    si HamD(G) = falso entonces devolver "¡No hay solución!"
    para cada e ∈ A hacer
        si HamD(⟨N, A \ {e}⟩) entonces A ← A \ {e}
    σ ← sucesión de nodos obtenida siguiendo el único ciclo que
        queda en G
```

Claramente, *Ham* requiere un tiempo polinómico si contamos con coste unitario todas las llamadas a *HamD*.

Consideremos dos problemas **A** y **B** tales que **A** $\leq_T^P$ **B**. Sea $p(n)$ un polinomio tal que el algoritmo de reducción para el problema **A** nunca requiere la resolución de más de $p(n)$ casos del problema **B** cuando se resuelve un caso de tamaño n, y tal que ninguno de esos casos es de tamaño mayor que $p(n)$. Este polinomio tiene que existir, porque en caso contrario el algoritmo de reducción requeriría un tiempo mayor que el polinómico, incluso contando las llamadas al algoritmo **B** con un coste unitario. Suponga ahora que existe un algoritmo *ResolverB* que es capaz de resolver el problema **B** en un tiempo que está en $O(t(n))$ para alguna función $t(n)$ asintóticamente no decreciente. Ahora podemos ejecutar el algoritmo de reducción

para resolver el problema **A**, llamando a *ResolverB* cuando sea necesario. Todo el tiempo que se invierta dentro de *ResolverB* estará en $O(p(n)t(p(n)))$ puesto que no serán necesarias más de $p(n)$ llamadas aplicadas a casos de tamaño $p(n)$ como máximo. Por tanto, el problema A se puede resolver en un tiempo que está en $O(p(n)t(p(n))+ q(n))$, donde $q(n)$ es un polinomio que tiene en cuenta el tiempo requerido por el algoritmo de reducción fuera de las llamadas a *ResolverB*. Considere ahora lo que sucede si $t(n)$ es un polinomio. En este caso, $p(n)t(p(n)) + q(n)$ es también un polinomio porque las sumas, productos y composiciones de polinomios son polinomios. Por tanto, tenemos el siguiente teorema fundamental.

Teorema 12.5.6 *Considere dos problemas* **A** *y* **B**. *Si* $\mathbf{A} \leq_T^p \mathbf{B}$ *y si* **B** *se puede resolver en tiempo polinómico, entonces* **A** *también se puede resolver en tiempo polinómico.*

Demostración: Esto se sigue inmediatamente de lo expuesto más arriba.

En particular, sabemos por el Teorema 12.5.5 que existe un algoritmo de tiempo polinómico para hallar ciclos hamiltonianos si y solo si existe un algoritmo de tiempo polinómico para decidir si un grafo es o no hamiltoniano. Por definición, esto último es equivalente a decir que **HAMD** $\in$ **P** puesto que **HAMD** es un problema de decisión. Por tanto, la cuestión consistente en si es o no posible hallar ciclos hamiltonianos en un tiempo polinómico es equivalente a una cuestión que concierne a la pertenencia a P a pesar de que la clase P solo está definida para problemas de decisión. Esto es típico de muchos problemas interesantes, que son polinómicamente equivalentes a problemas de decisión similares. De estos problemas decimos que son *reducibles a decisión*. Precisamente por ser frecuente la posibilidad de una reducción a decisión, no es grave la falta de generalidad cometida al definir P y NP como clases de problemas de decisión. Cuando uno está interesado en un problema que no es de decisión, lo más probable es que sea posible hallar un problema de decisión similar, que sea polinómicamente equivalente. El problema que nos interesa se puede resolver en tiempo polinómico si y solo si el problema de decisión correspondiente se encuentra en P. Véanse los problemas 12.43, 12.45, 12.46 y 12.47 para más ejemplos.

La restricción a problemas de decisión nos permite presentar una noción simplificada de reducción polinómica.

Definición 12.5.7 *Sean X e Y dos problemas de decisión definidos sobre un conjunto de casos I y J, respectivamente. El problema X es* polinómicamente reducible muchos a uno *al problema Y si existe una función $f: I \rightarrow J$ computable en tiempo polinómico tal que $x \in X$ si y solo si $f(x) \in Y$ para todo caso $x \in I$ del problema X. Esto se denota $X \leq_m^p Y$ y la función f se denomina* función de reducción. *Cuando $X \leq_m^p Y$ e $Y \leq_m^p X$, se dice que X e Y son* polinómicamente equivalentes muchos a uno, *se denota $X \equiv_m^p Y$.*

En otras palabras, la función de reducción hace corresponder todos los casos-si del problema X con casos-si del problema Y, y todos los casos no del problema X con casos-no del problema Y; véase la figura 12.7. Observe que es condición necesaria para que la función f sea computable en tiempo polinómico que el tamaño de $f(x)$ esté acotado superiormente por algún polinomio en el tamaño de x para todo $x \in I$. Las reducciones muchos a uno son herramientas útiles para establecer reducciones de Turing: para decidir si $x \in X$, basta calcular $y = f(x)$ y preguntar si y pertenece o no a Y. Entonces se tiene el teorema siguiente.

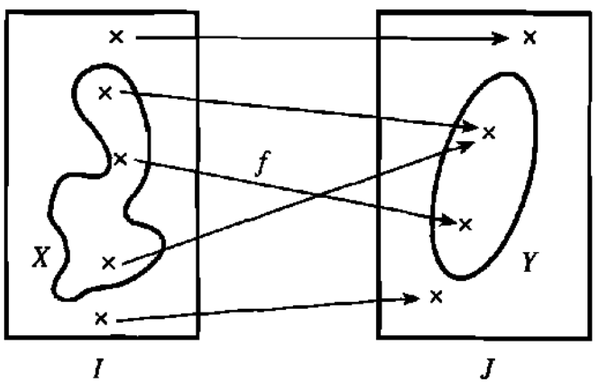


Figura 12.7 Reducción muchos a uno

Teorema 12.5.8 *Si X e Y son dos problemas de decisión tales que $X \leq_m^p Y$, entonces $X \leq_T^p Y$.*

Demostración: Imagine que las soluciones del problema Y se pueden obtener con un coste unitario mediante una llamada a *Decidir*Y y sea f la función de reducción existente entre X e Y, computable en un tiempo polinómico. Considere el algoritmo siguiente.

```
función DecidirX(x)
    y ← f(x)
    si DecidirY(y)    entonces devolver verdadero
                      sino devolver falso
```

Por definición de función de reducción, este algoritmo resuelve el problema X. Dado que la función de reducción es computable en un tiempo polinómico, resuelve el problema X en un tiempo polinómico, contando la llamada a *Decidir*Y con un coste unitario.

Este teorema resulta tan útil que es frecuente demostrar que $X \leq_m^p Y$ cuando realmente se necesita demostrar que $X \leq_T^p Y$. Sea consciente de que el recíproco de este teorema no es cierto en general: es posible para dos problemas de decisión X e Y que sea $X \leq_T^p Y$ y sin embargo $X \leq_m^p Y$; véanse los problemas 12.38, 12.39 y 12.40.

Considere, por ejemplo, el problema del viajante de comercio. Un caso de este problema consiste en un grafo con costes en las aristas. El problema de optimización, denotado como **TSP**, consiste en encontrar un recorrido del grafo que empiece y termine en algún nodo, después de haber visitado todos los demás nodos

exactamente una vez, y cuyo coste sea el mínimo posible; si ese recorrido no existe, la respuesta no está definida. Para definir un caso del problema de *decisión* **TSPD**, se proporciona una cota L además del grafo: la cuestión es decidir si existe o no un recorrido válido cuyo coste total no supere a L. El problema 12.47 pide demostrar que este problema es reducible a decisión: **TSP** $\equiv_T^P$ **TSPD**. Ahora demostramos que el problema del ciclo hamiltoniano es polinómicamente reducible al problema del viajante. De hecho, estos problemas son polinómicamente equivalentes, pero la reducción en el otro sentido es más difícil.

Teorema 12.5.9 **HAMD** $\leq_m^P$ **TSPD**

Demostración: Sea $G = \langle N, A\rangle$ un grafo con n nodos. Deseamos decidir si es hamiltoniano. Definimos $f(G)$ como el caso de **TSPD** que es el grado completo $H = \langle N, N \times N\rangle$, la función de coste

$$c(u,v) = \begin{cases} 1 & \text{si } \{u,v\} \in A \\ 2 & \text{en caso contrario} \end{cases}$$

y la cota $L = n$. Todo ciclo hamiltoniano de G se traduce en un recorrido de H que tiene un coste exactamente igual a n. Por otra parte, si no hay ciclos hamiltonianos en G,entonces todo recorrido de H debe utilizar al menos una arista de coste 2, y por tanto el coste total tiene que ser al menos $n + 1$. Por tanto, G es un caso si de **HAMD** si y solo si $f(G) = \langle H, c, L\rangle$ es un caso si de **TSPD**. Esto demuestra que **HAMD** $\leq_m^P$ **TSPD** porque la función f es fácil de calcular en tiempo polinómico.

12.5.3 Problemas NP-completos

Tal como vimos, la pregunta fundamental en lo que concierne a las clases P y NP es si la inclusión P $\subseteq$ NP es o no estricta. ¿Existe algún problema que admita un sistema de pruebas eficiente pero tal que sea inherentemente difícil descubrir certificados en el caso peor? Nuestra intuición y experiencia, nos llevan a creer que en general es más difícil descubrir una demostración que verificarla: de no ser así, el progreso en matemáticas sería mucho más rápido. Esta intuición se traduce en la conjetura de que P $\neq$ NP. Quienes trabajan en la teoría de la complejidad sufren una notable frustración al no poder confirmar ni rechazar esta conjetura. Si realmente existe una demostración sencilla de que P $\neq$ NP, ¡no ha sido sencillo encontrarla!

Por otra parte, uno de los grandes éxitos de esta teoría es la demostración de que existe un elevado número de problemas prácticos en NP tales que si cualquiera de ellos estuviera en P entonces todo NP sería igual a P. La evidencia que apoya la conjetura de que P $\neq$ NP presta entonces credibilidad al punto de vista de que ninguno de estos problemas se puede resolver mediante un algoritmo de tiempo polinómico en el caso peor. Estos problemas se denominan NP-completos. Para ser NP-completo, un problema de decisión tiene que pertenecer a NP y tiene que ser posible reducir polinómicamente cualquier otro problema de NP a ese problema.

Definición 12.5.10 *Un problema de decisión X es NP-completo si*
◊ $X \in NP$, *y*
◊ $Y \leq_T^p X$ *para todo problema* $Y \in NP$.

Algunos autores sustituyen la segunda condición por $Y \leq_m^p X$ o por otras clases de reducciones. No se sabe si esto da lugar a una clase verdaderamente diferente de problemas NP-completos.

¿Qué sucedería si algún problema NP-completo X se pudiera resolver en un tiempo polinómico? Considere cualquier otro problema $Y \in$ NP. Se tiene que $Y \leq_T^p X$ por definición, puesto que X es NP-completo. Por tanto, Y también se puede resolver en un tiempo polinómico, por el Teorema 12.5.6. Por tanto, cualquier problema de NP pertenece a P, lo cual implica que NP $\subseteq$ P. Peor sabemos que P $\subseteq$ NP, y por tanto P = NP. Esto demuestra que si algún problema NP-completo se puede resolver en tiempo polinómico, entonces todos los problemas en NP también. A la inversa, ningún problema NP-completo se puede resolver en tiempo polinómico si asumimos que P $\neq$ NP.

¿Cómo se puede demostrar que un problema es NP-completo? Si ya tenemos un conjunto de problemas de los cuales se ha demostrado que son NP-completos, entonces resulta útil el teorema siguiente.

Teorema 12.5.11 *Sea X un problema NP-completo. Consideremos un problema de decisión* $Z \in NP$ *tal que* $X \leq_T^p Z$. *Entonces Z también es NP-completo.*

Demostración: Para ser NP-completo, Z tiene que satisfacer las dos condiciones, de la Definición 12.5.10. La primera es que $Z \in$ NP, lo cual está en el enunciado del teorema. Para la segunda condición, considérese un $Y \in$ NP arbitrario. Dado que X es NP-completo y que $Y \in$ NP, se sigue que $Y \leq_T^p X$. Por hipótesis, $X \leq_T^p Z$. Por la transitividad de las reducciones polinómicas, $Y \leq_T^p Z$, que es lo que teníamos que demostrar para determinar que Z pertenece a NP.

Para demostrar que Z es NP-completo, seleccionamos un problema adecuado del grupo de problemas de los que ya sabemos que son NP-completos, y demostramos que es reducible polinómicamente a Z, bien muchos a uno o bien en el sentido de Turing. También hay que demostrar que $Z \in$ NP, mostrando un sistema de pruebas eficiente para Z. De esta manera se han enumerado varios miles de problemas NP-completos. Por ejemplo, basta probar que el problema del ciclo hamiltoniano es NP-completo para concluir, a la vista del Teorema 12.5.9, que el problema del viajante es también NP-completo.

Todo esto está muy bien cuando el proceso está en marcha, puesto que cuantos más problemas haya en el grupo, más probable es que podamos encontrar uno que se pueda reducir sin demasiada dificultad a algún problema nuevo. Lo difícil, por supuesto, es poner en marcha el sistema. ¿Qué podríamos hacer al principio,

cuando el conjunto de problemas es vacío, para demostrar por vez primera que algún problema en particular es NP-completo? Este es el *tour de force* que Steven Cook y Leonid Levin consiguieron realizar independientemente al principio de los años 70, abriendo camino para toda la teoría de la NP-completitud. La demostración completa de este resultado es técnicamente difícil; en el resto de la sección, daremos una visión somera de las ideas fundamentales subyacentes. Recuerde que las fórmulas booleanas se revisan en la Sección 1.4.1.

Definición 12.5.12 *Una fórmula booleana es satisfactoria si existe al menos una forma de asignar valores a sus variables de tal modo que resulte ser verdadera. Denotaremos mediante* **SAT** *el problema de decidir, dada una fórmula booleana, si es o no satisfactoria.*

Por ejemplo, $(p \vee q) \Rightarrow (p \wedge q)$ es satisfactoria porque da lugar a ser *verdadero* si asignamos el valor *verdadero* tanto a p como a q. Esta fórmula es viable a pesar de que existen otras asignaciones de valor para las variables que la hacen *falsa*, como p = *verdadero* y q = *falso*. Por otra parte, $(\neg p) \wedge (p \vee q) \wedge (\neg q)$ no es viable porque sigue siendo *falsa* independientemente de los valores que asignemos a p y a q. ¡Compruébelo!

En principio, es posible decidir si una fórmula booleana es satisfactoria calculando su valor para todas las posibles asignaciones de sus variables booleanas. Sin embargo, esto no es práctico cuando el número n de variables booleanas implicadas es elevado, porque hay 2^n asignaciones posibles. No se conoce ningún algoritmo eficiente para resolver este problema. Por otra parte, toda asignación que supuestamente satisfaga una fórmula booleana es a la vez sucinta y fácil de verificar, lo cual muestra que **SAT** $\in$ **NP**.

Consideremos ahora un caso especial de fórmulas booleanas.

Definición 12.5.13 *Un* literal *es o bien una variable Booleana o bien su negación. Una* cláusula *es o bien un literal o bien una disyunción de literales. Una fórmula Booleana está en la* forma normal conjuntiva *(CNF) si es una cláusula o una conjunción de cláusulas. Está en la forma k-***CNF** *para algún entero positivo k si está formada por cláusulas, cada una de las cuales contiene como máximo k literales.*

Por sencillez de notación, es frecuente representar la disyunción (o) en esas fórmulas mediante el signo "+", y la conjunción (y) por simple yuxtaposición de los operandos, como si fuera una multiplicación aritmética; la negación suele denotarse mediante una barra horizontal sobre la variable afectada.

Considere, por ejemplo, las fórmulas siguientes

$$(p + \bar{q} + r)(\bar{p} + q + r)q\bar{r}$$

$$(p + qr)(\bar{p} + q(q + r))$$

$$(p \Rightarrow q) \Leftrightarrow (\bar{p} + q)$$

La primera fórmula consta de cuatro cláusulas. Está en 3-**CNF**, y por tanto en **CNF**, pero no en 2-**CNF**. La segunda fórmula no está en **CNF** puesto que ni $(p + qr)$ ni $(\overline{p} + q(q + r))$ son cláusulas. La tercera fórmula tampoco esta en **CNF**, puesto que contiene operadores distintos de la conjunción, disyunción y negación.

Definición 12.5.14 **SAT-CNF** *es la restricción del problema* **SAT** *a fórmulas Booleanas en* **CNF**. *Para todo k positivo,* **SAT-*k*-CNF** *es la restricción de* **SAT-CNF** *a fórmulas Booleanas en k-*CNF.

Claramente, esos problemas están en NP. Se conoce un algoritmo eficiente para resolver **SAT-2-CNF**, pero incluso **SAT-3-CNF** es intratable en nuestro estado actual de conocimientos. Esto no es sorprendente, por cuanto veremos en la Sección 12.5.4 que este último problema es NP-completo.

La relevancia de las fórmulas booleanas en el contexto de los problemas NP-completos surge de su capacidad para simular algoritmos. Considere un problema arbitrario de decisión que se pueda resolver mediante un algoritmo A de tiempo polinómico. Supongamos que el tamaño de los casos se mide en bits. A todo entero n le corresponde una fórmula booleana $\Psi_n(A)$ en **CNF** que se puede obtener eficientemente. Esta fórmula contiene un elevado número de variables, entre las cuales $x_1, x_2, \ldots x_n$ corresponde de forma natural a los bits de casos de tamaño n para A. La fórmula booleana se construye de tal modo que exista una forma de satisfacerla, seleccionando los valores de las otras variables booleanas, si y solo si el algoritmo A admite el caso correspondiente al valor booleano de las variables x. Por ejemplo, el algoritmo A admite el caso 10010 si y solo si la fórmula $x_1\overline{x}_2\overline{x}_3x_4\overline{x}_5\Psi_5(A)$ es viable.

La demostración de que esta fórmula booleana existe y se puede construir eficientemente supone dificultades técnicas que van más allá del alcance de este libro. Nos contentamos con mencionar que la fórmula $\Psi_n(A)$ contiene entre otras cosas una variable booleana distinta b_{it} para cada bit i de almacenamiento que pueda necesitar emplear el algoritmo A para resolver un caso de tamaño n, y para cada unidad de tiempo t que requiera este cálculo. Una vez que están determinadas las variables $x_1, x_2, \ldots x_n$, las cláusulas de $\Psi_n(A)$ obligan a las demás variables booleanas a simular la ejecución paso a paso del algoritmo para el caso correspondiente.

Considere ahora cualquier problema arbitrario $Y \in$ NP cuyo espacio de prueba y cuyo sistema eficiente de prueba son Q y F respectivamente. Supongamos sin pérdida de generalidad que existe un polinomio $p(n)$ tal que para todo $y \in Y$ existe un certificado $q \in Q$ cuya longitud es exactamente $p(n)$, donde n es la longitud de y. Suponiendo que podamos resolver casos de **SAT-CNF** con un coste unitario, deseamos decidir eficientemente si $y \in Y$ para cualquier caso dado y. Para esto, considere el algoritmo A_y, cuyo propósito específico es verificar si su entrada es un certificado de que $y \in Y$. En otras palabras, $A_y(q)$ devuelve *verdadero* si y solo si $\langle y,q \rangle \in F$. Esto se puede hacer eficientemente por la hipótesis de que el sistema

de prueba de F es eficiente. Por definición, la fórmula booleana $\Psi_{p(n)}(A_y)$ es satisfactoria si y solo si existe un q de longitud $p(n)$ tal que A_y admite la entrada q. Por definición del sistema de pruebas, esto es equivalente a decir que $y \in Y$. Por tanto, basta decidir si es o no viable $\Psi_{p(n)}(A_y)$ para sabe si y pertenece o no a Y. Esto muestra la forma de reducir un caso arbitrario del problema Y a la viabilidad de una fórmula booleana en CNF, y por tanto $Y \leq_m^p$ **SAT-CNF**. Concluimos que $Y \leq_T^p$ **SAT-CNF** para todo problema Y perteneciente a NP. Al recordar que **SAT-CNF** está a su vez en NP, obtenemos el siguiente teorema fundamental.

Teorema 12.5.15 (Cook) **SAT-CNF** es NP-completo.

Pertrechados con este primer resultado de NP-completitud, podemos ya aplicar el Teorema 12.5.11 para demostrar la NP-completitud de otros problemas.

12.5.4 Algunas demostraciones de NP-completitud

Acabamos de ver que **SAT-CNF** es NP-completo. Sea Z algún otro problema de decisión en NP. Para demostrar que Z es también NP-completo, el Teorema 12.5.11 resulta aplicable, y solo necesitamos demostrar que **SAT-CNF** $\leq_T^p Z$. En lo sucesivo, para demostrar que algún otro W en NP es NP-completo, tenemos la opción de demostrar que **SAT-CNF** $\leq_T^p W$ o bien que $Z \leq_T^p W$. Tenga cuidado para no proceder a la inversa: el problema del que ya se sabe que es NP-completo es el que debemos de reducir al nuveo problema, y no al revés. Ilustraremos este principio con algunos ejemplos.

Teorema 12.5.16 **SAT** es NP-completo.

Demostración: Ya sabemos que **SAT** está en NP. Dado que **SAT-CNF** es el único problema del que sabemos que es NP-completo hasta el momento, tenemos que demostrar que **SAT-CNF** $\leq_T^p$ **SAT** para aplicar el Teorema 12.5.11. Esto es inmediato, por cuanto las fórmulas booleanas en **CNF** son un caso especial de las fórmulas booleanas generales, y es fácil decidir, dada una fórmula booleana, si está o no en **CNF**. Por tanto, todo algoritmo capaz de resolver **SAT** eficientemente puede utilizarse directamente para resolver **SAT-CNF**.

Teorema 12.5.17 **SAT-3-CNF** es NP-completo.

Demostración: Ya sabemos que **SAT-3-CNF** está en NP. Como ya conocemos dos problemas distintos que son NP-completos, tenemos la opción de demostrar o bien que **SAT-CNF** $\leq_T^p$ **SAT-3-CNF** o bien que **SAT** $\leq_T^p$ **SAT-3-CNF**. Vamos a demostrar lo primero, procediendo por reducción muchos a uno: demostramos que **SAT-CNF** $\leq_m^p$ **SAT-3-CNF**. Considérese una fórmula booleana arbitraria Ψ que sea viable. Tenemos que construir eficientemente una fórmula $\xi = f(\Psi)$ en **3-CNF** que sea viable si y solo si Ψ es viable. Considere primero el caso en que Y contiene una sola cláusula, que es una disyunción de k literales.

◊ Si $k \leq 3$, sea $\xi = \Psi$, que ya está en **3-CNF**.
◊ Si $k = 4$, sean ℓ_1, ℓ_2, ℓ_3, y ℓ_4 literales tales que Ψ es $\ell_1 + \ell_2 + \ell_3 + \ell_4$. Sea u una nueva variable booleana. Se toma

$$\xi = (\ell_1 + \ell_2 + u)(\overline{u} + \ell_3 + \ell_4).$$

Obsérvese que si al menos uno de los ℓ_i es *verdadero*, entonces Ψ es *verdadero*, y es posible seleccionar un valor de verdad para u tal que ξ también sea *verdadero*. A la inversa, si todos los ℓ_i son *falsos*, entonces Ψ es *falso* y ξ es *falso* sea cual fuere el valor de verdad seleccionado para u.

◊ Más generalmente, si $k \geq 4$, sean ℓ_1, ℓ_2, ..., ℓ_k los literales tales que es $\Psi = \ell_1 + \ell_2 + \ldots + \ell_k$. Sean $u_1, u_2, \ldots, u_{k-3}$, nuevas variables booleanas. Se toma

$$\xi = (\ell_1 + \ell_2 + u_1)(\overline{u}_1 + \ell_3 + u_2)\cdots(\overline{u}_{k-3} + \ell_{k-1} + l_k)$$

Una vez más, dados unos valores cualesquiera de verdad para los ℓ_i, Ψ es *verdadero* si y solo si ξ es viable con una selección adecuada de asignaciones para los u_i.

Si la fórmula Ψ consta de varias cláusulas, cada una de ellas se trata independientemente —empleando distintas variables u para cada cláusula— y se forma la conjunción de todas las expresiones en **3-CNF** obtenidas así. Por ejemplo, si

$$\Psi = (p + \overline{q} + r + s)(\overline{r} + s)(\overline{p} + s + \overline{x} + v + \overline{w})$$

entonces obtenemos

$$\xi = (p + \overline{q} + u_1)(\overline{u}_1 + r + s)(\overline{r} + s)(\overline{p} + s + u_2)(\overline{u}_2 + \overline{x} + u_3)(\overline{u}_3 + v + w)$$

Dado que cada cláusula se "traduce" con un conjunto diferente de variables u, y dado que la única manera de satisfacer Y es satisfacer todas y cada una de sus cláusulas con una misma asignación de verdad para las variables booleanas, toda asignación satisfactoria para Ψ da lugar a una para ξ, y viceversa. En otras palabras, Y es viables si y solo ξ lo es también. Pero x está en **3-CNF**. Esto muestra la forma de transformar eficientemente una fórmula CNF arbitraria en una en **3-CNF** de tal manera que su viabilidad se mantenga. Por tanto, $SAT - CNF \leq_m^p SAT - 3 - CNF$, lo cual completa la demostración de que **SAT-3-CNF** es NP-completo.

Definición 12.5.18 *Sea G un grafo no dirigido, y sea n un entero. Un* coloreado *de G es una asignación de colores a los nodos de G de tal manera que dos nodos cualesquiera que estén conectados por una arista sean siempre de diferentes colores. Si no utiliza más de k colores diferentes, entonces es un* coloreado *k. El menor k tal que existe un coloreado k del grafo se llama* número cromático *del grafo, y cualquier coloreado k de estos será un* coloreado óptimo. *Definimos los cuatro problemas siguientes:*

◊ 3COL: Dado un grafo G, ¿se puede colorear G con 3 colores?
◊ COLD: Dado un grafo G y un entero k, ¿se puede colorear G con k colores?
◊ COLO: Dado un grafo G, hallar el número cromático de G.
◊ COLC: Dado un grafo G, hallar un coloreado óptimo de G.

Los problemas 12.43 y 12.44 piden demostrar que estos problemas son polinómicamente equivalentes: cualquiera de ellos se puede resolver en un tiempo polinómico si y solo si se pueden resolver todos. Dado que estamos a punto de demostrar que 3COL es NP-completo, esta es una evidencia de que los cuatro problemas son difíciles.

Teorema 12.5.19 3COL es NP-completo.

Demostración: Es fácil ver que **3COL** está en NP, puesto que cualquier supuesto tricoloreado se puede verificar eficientemente. Para mostrar que **3COL** es NP-completo, demostraremos esta vez que **SAT-3-CNF** $\leq_m^p$ **3COL.** Dada una fórmula booleana Ψ en **3-CNF**, tenemos que construir eficientemente un grafo G que se puedan pintar con tres colores si y solo si Ψ es satisfactoria. Esta reducción es considerablemente más complicada que las vistas hasta el momento.

Supongamos por sencillez que todas las cláusulas de la fórmula Ψ contienen exactamente tres literales (véase el problema 12.54). Sea k el número de cláusulas de Ψ. Supongamos además, sin pérdida de generalidad, que las variables booleanas que aparecen en Ψ son $x_1, x_2, \ldots, x_t$. El grafo G que vamos a construir contiene $3 + 2t + 6k$ nodos y $3 + 3t + 12k$ aristas. Tres nodos especiales de este grafo están enlazados en un *triángulo de control* que se muestra en la parte superior de la figura 12.8: los llamaremos V, F y C. Dado que cada uno de ellos está enlazado con los otros dos, tienen que ser de colores distintos en todo coloreado válido de este grafo. Cuando llegue el momento de pintar G en tres colores, imagine que los colores asignados a V y F representan los valores booleanos *verdadero* y *falso* respectivamente. Diremos que un nodo tiene el color *verdadero* si su color es el mismo que el de V, y de forma análoga hablamos de nodos de color *falso*. Todo nodo que posea el color *verdadero* o *falso* se llamará *nodo de verdad*.

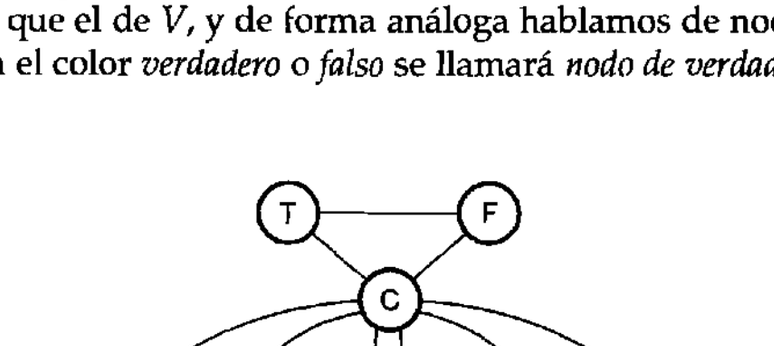


Figura 12.8 ***Representación de tres variables booleanas mediante un grafo***

Para cada variable booleana X_i de Ψ, el grafo contiene dos nodos y_i y z_i enlazados entre sí y con el nodo de control C. En cualquier tricoloreado válido de G, esto obliga a que y_i tenga o bien el color *verdadero* o bien el *falso*, y z_i tendrá que ser del color complementario. Piense que el color del nodo Y_i es la asignación de verdad de la variable booleana x_i, así que el color del nodo z_i corresponde al valor de verdad de x_i. Podemos pensar que

los y_i y z_i corresponden a los literales de la fórmula booleana. Por ejemplo, la figura 12.8 muestra la parte del grafo que hemos construido hasta el momento si la fórmula tiene tres variables.

Aún quedan por añadir 6 nodos y 12 aristas por cada cláusula de Ψ. Esos se añaden de tal modo que el grafo se pueda colorear con tres colores si y solo si la selección de colores para $y_1, y_2, \ldots, y_t$ corresponde a la asignación de valores booleanos a $x_1, x_2, \ldots, x_i$ que satisface todas las cláusulas. Esto se logra mediante el *artilugio* ilustrado en la figura 12.9. Decimos que un artilugio está *enlazado* con los nodos *a*, *b* y *c* si estos son los extremos de los nodos marcados con 1, 2 y 3. Todo artilugio está conectado también directamente con los nodos **V** y **C** mediante las otras dos aristas de la figura. Probando las ocho posibilidades, se puede demostrar que si el artilugio solo está conectado a nodos de verdad, entonces se puede pintar con los colores asignados al triángulo de control si y solo si está enlazado con al menos un nodo cuyo color sea *verdadero*. Por tanto, el artilugio puede utilizarse para simular la disyunción de los tres literales representados por los nodos con los cuales está unido. Para completar el grafo, basta con incluir una copia del artilugio por cada cláusula de Ψ. Cada artilugio está enlazado con los nodos seleccionados desde los y_i y z_i de tal manera que corresponda a los tres literales de la cláusula considerada. Todo tricoloreado válido del grafo proporciona una asignación de verdad para la fórmula booleana, y viceversa. Por tanto, el grafo se puede pintar con los tres colores si y solo si Ψ es viable. Dado el grafo que se puede construir eficientemente comenzando desde cualquier fórmula booleana Ψ en **3-CNF**, concluimos que $SAT - 3 - CNF \leq_m^p 3COL$ y por tanto que **3COL** es NP-completo.

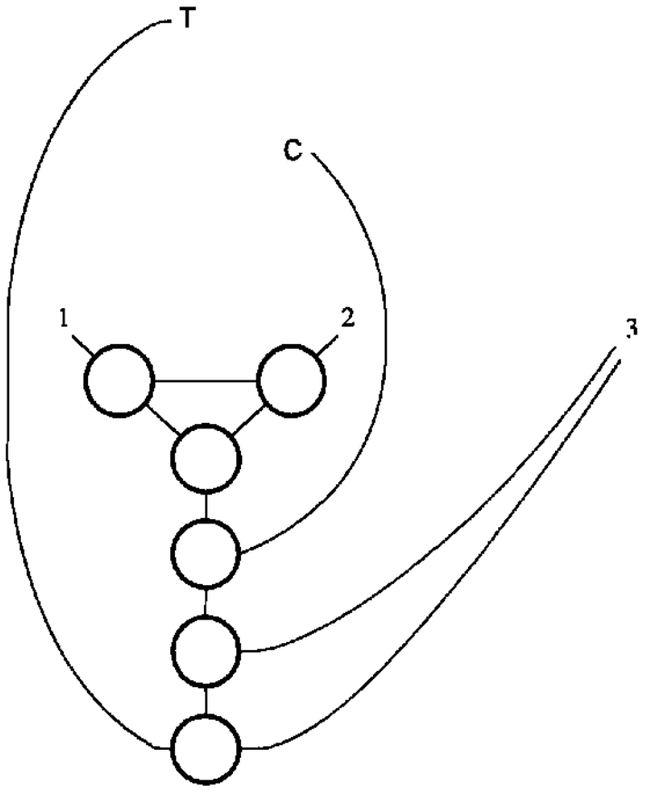


Figura 12.9 Un artilugio

12.5.5 Problemas NP-difíciles

No es necesario demostrar que un problema pertenece a NP para ofrecer evidencias de que no es posible resolverlo eficientemente. Diremos que un problema *X*

es de *NP-difícil* si existe un problema NP-completo Y que se pueda reducir Turing polinómicamente a él: $Y \leq_T^p X$. Por definición de reducciones polinómicas, todo algoritmo de tiempo polinómico para X daría lugar a uno para Y. Dado que Y es NP-completo, esto implicaría que P = NP, en contra de las creencias actuales. Por tanto, ningún problema NP-difícil se puede resolver en tiempo polinómico en caso peor, suponiendo que P ≠ NP.

Hay varias razones para estudiar la NP-dificultad en lugar de la completitud NP. En particular, los problemas de NP-dificultad no tienen por qué ser problemas de decisión. Considere por ejemplo los problemas de coloreado de grafos enunciado en la Definición 12.5.18. Es evidente que todo algoritmo eficiente para hallar el coloreado óptimo de un grafo (**COLC**) o para determinar el número cromático de un grafo (**COLO**) se puede utilizar para determinar eficientemente si un grafo se puede pintar con tres colores (**3COL**). Simbólicamente, $3COL \leq_T^p COLO \leq_T^p COLC$. Dado que **3COL** es NP-completo, se sigue que tanto **COLO** como **COLC** son problemas NP-difíciles, aun cuando no sean NP-completos, porque no son problemas de decisión. Veremos muchos problemas NP-difíciles que no son completos por esta razón en el Capítulo 13.

La noción de NP-dificultad también es interesante para los problemas de decisión. Existen problemas de decisión de los cuales se sabe que son NP-difíciles, pero de los cuales se cree que no están en NP, y por tanto no son NP-completos. Considere por ejemplo el problema **COLE** del coloreado *exacto*: dado un grafo G y un entero k, ¿puede pintarse G con k colores pero no con menos? Una vez más, es evidente que $3COL \leq_T^p COLE$ porque un grafo es tricoloreable si y solo si su número cromático es 0, 1, 2 o 3. Dado que **3COL** es NP-completo, concluimos que el problema de coloreado exacto de grafos es NP-difícil. Sin embargo, este problema de decisión no parece estar en NP. Aun cuando todo coloreado válido de G con k colores se puede emplear como certificado sucinto de que se puede pintar con k colores, es difícil imaginar qué forma puede tener un certificado sucinto de que G *no* se puede pintar con menos colores, y existen fuertes razones teóricas para creer que estos certificados no existen en general.

Finalmente, la NP-dificultad suele ser lo único que realmente deseamos confirmar. A no ser que resulte que P = NP, en la práctica no es muy útil saber que un problema dado pertenece a NP. Por tanto, aun cuando el problema considerado sea un problema de decisión, e incluso si es razonable esperar que esté en NP, ¿para qué vamos a desperdiciar tiempo y dinero construyendo un sistema de pruebas para él?

12.5.6 Algoritmos no deterministas

La clase NP se definió originalmente de forma bastante distinta, aun cuando las definiciones sean equivalentes. La definición clásica usa la noción de algoritmos no deterministas, que solo esbozaremos. Como dijimos antes, el nombre NP surgió de esta otra definición: representa la clase de problemas que se pueden resolver mediante un algoritmo No determinista en un tiempo Polinómico.

Aun cuando se pueden definir algoritmos no deterministas con respecto a problemas generales, una vez más nos concentraremos por sencillez en los problemas

de decisión. En este contexto, los algoritmos no deterministas finalizan su ejecución con **admitir** o **rechazar**, dos instrucciones especiales que se explican más adelante, o bien pueden quedar en un bucle infinito. Además, los algoritmos no deterministas pueden utilizar una instrucción especial

seleccionar n **entre** i **y** j

cuyo efecto es asignar a n algún valor entero entre i y j, ambos inclusive. El valor real que se le asigne a n no queda especificado por el algoritmo, ni está sujeto a las leyes de la probabilidad. Por tanto, los algoritmos no deterministas no deben de confundirse con los algoritmos probabilista.

El efecto del algoritmo está determinado por la existencia o inexistencia de sucesiones de elecciones no deterministas que dan lugar a una instrucción **admitir**. *No* nos concierne la forma en que estas sucesiones pudieran determinarse eficientemente, ni tampoco la forma en que pudiera determinarse su inexistencia. Por esta razón, los algoritmos no deterministas son únicamente una abstracción matemática que no se puede utilizar directamente en la práctica: nunca programaremos uno de estos algoritmos con la esperanza de poder ejecutarlo eficientemente en una computadora real. En particular, carecería de sentido sustituir las instrucciones **seleccionar** por selecciones probabilísticas tales como "$n \leftarrow$ *uniforme*$(i..j)$" porque en general la probabilidad de éxito sería infinitesimal. Por tanto, no nos preocupa el hecho consistente en que, como veremos, los algoritmos no deterministas pueden resolver problemas NP-completos en un tiempo polinómico.

Definición 12.5.20 *Un* cómputo admisible *del algoritmo es una sucesión de selecciones no deterministas que dan lugar a una instrucción* **admitir**. *El algoritmo* admite *la entrada x si posee al menos un cómputo admisible cuando se aplica a x; en caso contrario diremos que* rechaza *la entrada. El algoritmo* resuelve *el problema de decisión X si admite todos los $x \in X$ y rechaza todos los $x \notin X$.*
El tiempo *que requiere un algoritmo no determinista al aplicarlo a un caso que acepte se define como el tiempo más corto posible que puede ser requerido por un cómputo admisible del algoritmo, aplicado a este caso; el tiempo correspondiente a casos rechazados no está definido. Un algoritmo no determinista* corre en tiempo polinómico *si el tiempo que requiere en los casos admitidos está acotado por algún polinomio en el tamaño del ejemplar.*

Observe que un algoritmo no determinista admite la entrada x aun cuando solo tenga un cómputo admisible frente a muchos cómputos que lleven al rechazo. Observe también que no hay límite del tiempo que puede estar en funcionamiento un algoritmo no determinista en el tiempo polinómico si se toman "malas" decisiones no deterministas, o si se aplica a un caso rechazado; el algoritmo incluso puede caer en un bucle infinito en estos casos. Un cómputo puede ser arbitrariamente largo incluso en un caso admitido, siempre y cuando ese caso también admita un cómputo acotado polinómicamente.

Considere el siguiente ejemplo de algoritmo no determinista para decidir si un grafo es hamiltoniano. Se selecciona una sucesión de nodos de forma no determinista, con la esperanza de encontrar un ciclo hamiltoniano. Claramente, existe al menos una sucesión de decisiones que nos lleva a la admisión si y solo si existe un ciclo hamiltoniano. Por otra parte, no tendría sentido intentar ejecutar el algoritmo después de sustituir las decisiones no deterministas por otras probabilistas.

procedimiento *HamND*($G = <N, A>$)
{ Este algoritmo supone que el grafo contiene al menos tres nodos}
$n \leftarrow |N|$
sean los nodos $N = \{v_1, v_2, \ldots, v_n\}$
$S \leftarrow \varnothing$
$x \leftarrow v_1$
para $k \leftarrow 2$ **hasta** n **hacer**
  seleccionar i **entre** 2 **y** n
  si $i \in S$ **o** $\{x, v_i\} \notin A$ **entonces rechazar**
  $x \leftarrow v_i$
  $S \leftarrow S \cup \{x\}$
si $\{x, v_i\} \in A$ **entonces aceptar**
  sino rechazar

Consideremos ahora un problema arbitrario $X \in$ NP y sean Q y F su espacio de pruebas y su sistema de pruebas eficiente, respectivamente. Supongamos por sencillez que Q es el conjunto de todas las cadenas binarias. La relevancia de los algoritmos no deterministas es que, dado cualquier x, pueden seleccionar de forma no determinista un $q \in Q$ tal que $<x, q> \in F$, siempre y cuando exista ese q. Este que se puede seleccionar bit por bit, de acuerdo con una sucesión de elecciones binarias no deterministas. En cierto sentido, el algoritmo emplea su potencia no determinista para estimar un certificado de que $x \in X$ si es que existe alguno. Una vez estimado q, el algoritmo establece de forma determinista si $<x,q>$ pertenece o no a F y, de ser así, lo admite. Este algoritmo no determinista admite todos los casos-si, porque hay al menos una secuencia de opciones binarias no deterministas que llega a un certificado correcto, dando lugar a un cómputo admisible. Por otra parte, no se pueden admitir los casos-no, porque $<x,q> \notin F$ independientemente del q seleccionado cuando $x \notin X$. Además, este algoritmo no determinista funciona en un tiempo polinómico porque está garantizada la existencia de un certificado sucinto, y porque la comprobación de que $<x,q> \in F$ se puede llevar a cabo en un tiempo polinómico por definición de NP.

Formalmente, el algoritmo no determinista de tiempo polinómico para resolver el problema X es el siguiente. Obsérvese que nunca para en los casos-no, lo cual está permitido en la definición de tiempo polinómico para los algoritmos no deterministas.

procedimiento $XND(x)$
$q \leftarrow$ cadena binaria vacía
mientras $<x,q> \notin F$ **hacer**
    seleccionar b **entre** 0 **y** 1
    añadir el bit b a la derecha de q
admitir

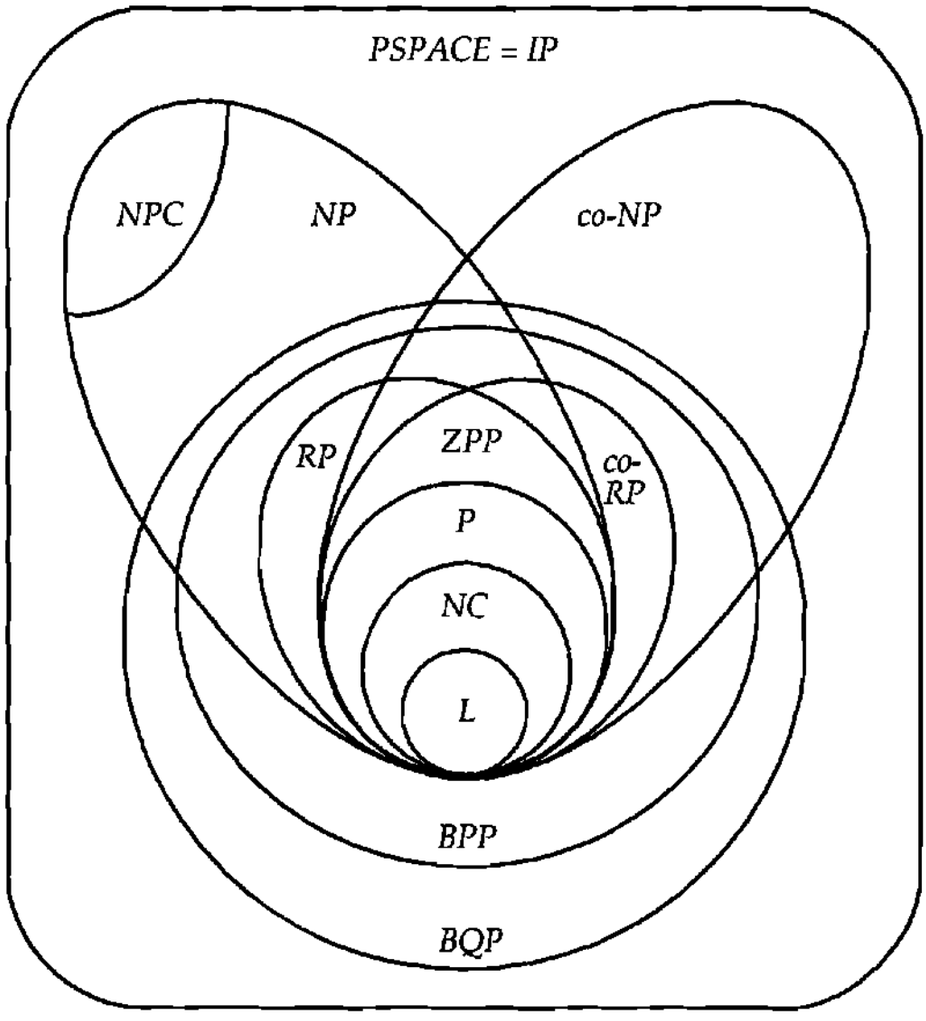
Recíprocamente, todo problema de decisión que se resuelva en tiempo polinómico mediante un algoritmo no determinista pertenece a NP. Para esto utilizamos el conjunto de todas las posibles sucesiones de elecciones no determinístas como espacio de prueba. El sistema de pruebas, F, se define como el conjunto de pares (x, s) tal que x es un caso-si y s es una sucesión de opciones no determinista según la cual el algoritmo no determinista admite la entrada x. Dado que el algoritmo no determinista funciona en un tiempo polinómico, al menos una de estas sucesiones admisibles es de longitud polinómica, así que existe al menos un certificado sucinto para cada caso si. Además, es fácil verificar en un tiempo determinista polinómico que un s dado es un certificado correcto de que $x \in X$: basta simular el algoritmo no determinista, salvo que la secuencia s se consulta para decidir de forma determinista la forma en que hay que proceder cada vez que llegamos a una elección no determinista.

Según esta discusión, vemos que la Definición 12.5.2 de NP es equivalente a decir que NP es la clase de problemas de decisión que se resuelven en tiempo polinómico mediante algoritmos no deterministas. Una vez más, esta fue la definición original de la que NP tomó su nombre.

12.6 UN ZOO DE CLASES DE COMPLEJIDAD

Una aproximación fructífera al estudio de la complejidad computacional es la basada en la noción de *clases de complejidad*. Una clase de complejidad consiste en el conjunto de todos los problemas de algún tipo (problemas de decisión, por ejemplo) que se pueden resolver empleando un modelo dado de cómputo, sin sobrepasar alguna cantidad dada de recursos. Por ejemplo, P es la clase de todos los problemas de decisión que se pueden resolver mediante algoritmos deterministas en un tiempo polinómico, y NP es la clase de todos los problemas de decisión que se pueden resolver mediante algoritmos no deterministas empleando un tiempo polinómico. Se han estudiado muchas otras clases de complejidad. Aquí nos hemos limitado a explorar superficialmente este extenso campo. La figura 12.10 resume la descripción posterior.

PSPACE es la clase de todos los problemas de decisión que se pueden resolver empleando como máximo un número polinómico de bits de almacenamiento. Más exactamente, un problema de decisión pertenece a PSPACE si existe un algoritmo A que lo resuelve y un polinomio $p(n)$ tal que la cantidad de espacio que necesita A para cualquier caso x no es mayor que $p(n)$ bits, donde n es el tamaño de x. Dado que cualquier algoritmo se puede transformar sin disminución



L y NPC denotan LOGSPACE y la clase de problemas NP-completos, respectivamente. Este diagrama es correcto siempre y cuando no existan problemas NP-completos en BQP, lo cual se cree pero no está demostrado. Se conjetura que todas las regiones son no vacías, pero solo se sabe con certeza que LOGSPACE y PSPACE son distintas.

Figura 12.10. Un zoo de clases de complejidad

significativa de su velocidad en uno que no necesite más espacio que el tiempo necesario, está claro que P $\subseteq$ PSPACE. Sin embargo, no sabemos si esta inclusión es propia o no. Esto es todavía más embarazoso que nuestra incapacidad para demostrar que P$\neq$NP porque NP $\subseteq$ PSPACE; véase el problema 12.60. Lo ha adivinado usted: no sabemos si esta última inclusión es o no propia. Sin embargo, hay cosas que sí sabemos acerca de PSPACE. Existen problemas de PSPACE a los cuales se pueden Turing reducir polinómicamente todos los demás problemas de PSPACE. Estos problemas *PSPACE-completos* se pueden resolver en tiempo polinómico si y solo si P=PSPACE. Un resultado sorprendente es que el no determinismo no añade mucha potencia de cálculo cuando el factor limitante es el espacio: PSPACE=NPSPACE donde NPSPACE es la versión no determinista de PSPACE; véase el problema 12.61.

Del mismo modo que se cree que PSPACE va más allá de P y de NP, se cree que LOGSPACE es más restrictivo que P. Esta es la clase de todos los problemas de decisión que se pueden resolver con una cantidad de espacio que no es mayor que al-

guna constante multiplicada por el logaritmo del tamaño del caso. Para que esta definición tenga sentido, suponemos que el caso se da en un almacenamiento de solo lectura, y contamos solamente el número de bits adicionales de almacenamiento con lectura y escritura que se necesitan para efectuar el cálculo. El problema 12.62 pide demostrar que LOGSPACE $\subseteq$ P, pero tenemos que admitir nuestra ignorancia acerca de si esta inclusión es o no propia. Sin embargo, sabemos *con certeza* que LOGSPACE está incluido estrictamente en PSPACE. Para resumir, sabemos que

$$\text{LOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$$

y al menos una de estas inclusiones es estricta. Se conjetura que lo son todas.

Si C es una clase de complejidad de problemas de decisión, denotamos mediante *co-C* la clase de problemas de decisión cuyos complementos están en C. En otras palabras, si I es un conjunto de casos y $X \subseteq I$ es un problema de decisión que pertenece a C, entonces $I \setminus X$ pertenece a co-C. Por ejemplo, el conjunto de fórmulas booleanas que *no* son viables y el conjunto de grafos que *no* poseen un ciclo hamiltoniano pertenecen a co-NP. Está claro que P es un subconjunto tanto de NP como de co-NP, y que NP y co-NP son ambos subconjuntos de PSPACE. La discusión que precede inmediatamente al Teorema 12.5.3 da credibilidad a la conjetura de que NP $\neq$ co-NP, pero esto no es algo que sepamos demostrar. Sin embargo, se sabe que NP = co-NP si y solo si existe un problema NP-completo en co-NP. No hay muchos problemas de los cuales se sepa que están en NP $\cap$ co-NP, y que sin embargo se crea que no están en P. Entre estos mencionaremos el conjunto de los números primos y el problema de decisión Turing equivalente polinómicamente a la factorización dado en el problema 12.48.

Los algoritmos probabilistas dan lugar a clases de complejidad probabilistas. Por razones técnicas, la definición formal de esas clases limita a los algoritmos probabilistas a lanzar monedas imparciales, en lugar de tener acceso a *uniforme*(a, b) para unos números reales arbitrarios a y b.

- ◊ Los algoritmos de Monte Carlo dan lugar a la clase BPP, que quiere decir *tiempo polinómico probabilístico con error acotado.* Un problema de decisión pertenece a BPP si existe un algoritmo probabilístico p-correcto que lo resuelve en tiempo polinómico para algún $p > \frac{1}{2}$. Tal como veíamos en la Sección 10.6.4, la probabilidad de error se puede reducir más allá de cualquier umbral deseado repitiendo el algoritmo un cierto número de veces y tomando la respuesta más frecuente.
- ◊ Los algoritmos de Las Vegas dan lugar a la clase ZPP, que quiere decir *tiempo polinómico probabilístia de error nulo.* Un problema de decisión pertenece a ZPP si existe un algoritmo probabilista que lo resuelve sin posibilidad de error, y con un tiempo esperado polinómico.
- ◊ Entre BPP y ZPP existe la clase RP, que quiere decir *tiempo polinómico aleatorio.* Un problema de decisión pertenece a RP si existe un algoritmo pro-

babilístico p-correcto que lo resuelva en tiempo polinómico para algún $p > 0$, y tal que se obtenga una respuesta correcta con certeza en todos los casos-no. Tal como vimos en la Sección 10.6.4, la probabilidad de error se puede reducir por debajo de cualquier umbral deseado de forma mucho más eficiente que en el caso de los algoritmos BPP generales. Se conjetura que RP ≠ co-RP.

Las siguientes relaciones entre clases de complejidad probabilistas y deterministas son válidas.

$$P \subseteq ZPP \subseteq RP \subseteq BPP \subseteq PSPACE$$

Además, los problemas 12.64 y 12.65 piden demostrar que RP ⊆ NP, y que ZPP = RP ∩ co-RP. Por otra parte, se cree que ni NP ni BPP son subconjuntos uno de otro.

Cuando definíamos NP dijimos que se podía pensar que se trataba de la clase de problemas de decisión para los cuaes un ser omnisciente nos pudiera convencer de la validez de todo caso-si mostrándonos un certificado sucinto cuya validez se pudiera verificar eficientemente, aun cuando nosotros no fuéramos capaces de hallar el certificado. Es natural extender esta noción para permitir *interacción* con el ser: él nos muestra algo, nosotros contestamos, nos responde, contestamos de nuevo y así sucesivamente. Un problema de decisión X pertenece a la clase IP, que quiere decir *interactivamente demostrable,* si el ser nos puede convencer de que $x \in X$ siempre que así sea, pero estamos casi seguros de cogerle en mintiendo si intenta convencernos de que $x \in X$ cuando la verdad es que no es así. Se requiere que toda la interacción necesite un tiempo acotado por un polinomio en el tamaño del caso, suponiendo que el ser responda instantáneamente. Véase el problema 12.66 para un ejemplo. Es evidente que NP ⊆ IP, puesto que la interacción podría consistir simplemente en que el ser nos mostrase un certificado NP. ¿Podría ocurrir que haya afirmaciones que se puedan demostrar interactivamente, pero solo al cabo de varias rondas? En otras palabras, ¿es estricta la inclusión NP ⊆ IP? Aun cuando no conocemos la respuesta con seguridad, podemos apostar con confianza a que así es, porque uno de los resultados recientes más sorprendentes en complejidad computacional es que IP = PSPACE.

Los algoritmos paralelos dan lugar a clases de complejidad paralelas. Aun cuando hay un gran número de ellas, solo mencionaremos las más populares. NC es la clase de problemas que se pueden resolver mediante algoritmos paralelos eficientes. Recuerde de la Sección 11.3 que esto significa que el problema se puede resolver en un tiempo polilogarítmico en un número polinómico de procesadores. (La clase NC se denominado así aludiendo a su inventor, Nicholas Pippenger: quiere decir Clase de Nicolás[1]). Es evidente que NC ⊆ P porque es sencillo simular polinómicamente muchos procesadores que funcionan durante un tiempo polilogarítmico en un único procesador y en un tiempo polinómico. También se sabe que LOGSPACE ⊆ NC. No se sabe que ninguna de las inclusiones sea estricta. La teoría de la NP-completitud tiene un paralelo muy próximo en el estudio de

NC: se sabe que los problemas *P -completos* están en P, pero están en NC si y solo si NC = P. El problema de decidir, dado un grafo dirigido $G = <N,A>$, y dos nodos $u,v \in N$, si existe o no un camino que va desde u a v, es P-completo, por ejemplo.

Finalmente, las computadoras cuánticas dan lugar a toda una nueva colección de clases de complejidad. Aun cuando no explicaremos en este libro que es una computadora cuántica, piense en un dispositivo que utiliza el principio de superposición de la mecánica cuántica para propósitos de computación. Esto permite un paralelismo masivo en un único dispositivo físico. La computación cuántica es reminiscente de los algoritmos probabilistas clásicos, salvo que en algunas ocasiones las "probabilidades" parecen volverse negativas gracias a un fenómeno conocido con el nombre de interferencia cuántica. No se sabe si las computadoras cuánticas son realmente más potentes que las máquinas probabilistas. Hay fuertes evidencias de que lo son, porque la factorización de enteros grandes se puede efectuar en tiempo polinómico mediante computadoras cuánticas, mientras que se cree que es una tarea intratable empleando máquinas clásicas. Por otra parte, existen fuertes evidencias de que las computadoras cuánticas no pueden resolver problemas NP-completos en tiempo polinómico. Además, las computadoras cuánticas siguen siendo algo irreal: van más allá del alcance de la tecnología actual. ¿Pero por cuánto tiempo? De todas las clases de complejidad cuántica mencionaremos solo una: BQP es la clase de problemas de decisión que se pueden resolver en un tiempo polinómico en una computadora cuántica de tal forma que se obtiene la respuesta correcta con una probabilidad que es al menos p en todos los casos, para algún $p > \frac{1}{2}$. Se sabe que BPP $\subseteq$ BQP $\subseteq$ PSPACE; se cree que ambas inclusiones son estrictas, pero no se ha llegado a demostrar.

12.7 PROBLEMAS

Problema 12.1. Demostrar por inducción matemática sobre la altura que un árbol binario de altura k tiene como máximo 2^k hojas. Concluir que todo árbol binario con t hojas tiene que tener una altura que sea como mínimo $\lceil \lg t \rceil$.

Problema 12.2. Considere un entero positivo k. Sea $t = \lceil \lg k \rceil$ y $l = k - 2^t$. Demostrar que $h(k) = kt + 2l$, donde $h(k)$ es la función que se utilizaba en la demostración del Teorema 12.2.1. Dar una interpretación intuitiva de esta fórmula en el contexto de la altura media de un árbol con k hojas.

Problema 12.3. Dar los árboles de decisión correspondientes a los algoritmos de ordenación por selección (Sección 2.4) y por fusión (Sección 7.4.1), y *quicksort* (Sección 7.4.2) para el caso de tres elementos. En los dos últimos casos, detener las llamadas recursivas cuando solo quede un elemento por "ordenar".

Problema 12.4. Dar un árbol de decisión válido para ordenar cuatro elementos.

Problema 12.5. Dar un árbol de decisión válido para determinar la mediana de cinco elementos. Obsérvese que solo tiene 5 veredictos distintos, pero muchas más hojas.

Problema 12.6. Dar las fórmulas exactas para el número de comparaciones que se efectúan en el caso peor por parte de los algoritmos de ordenación por selección e in-

serción, para ordenar n elementos. ¿Cómo se comportan estos algoritmos en comparación con la cota inferior de teoría de la información, $\lceil \lg 50! \rceil$, cuando $n = 50$?

Problema 12.7. Demostrar que si n es una potencia de 2, *ordenar-fusión* hace $n \lg n - n + 1$ comparaciones en el caso peor, para ordenar n elementos. ¿Qué relación tiene esto con la cota inferior de teoría de la información, $\lceil \lg n! \rceil$? Hallar la menor potencia de 2 tal que *ordenar-fusión* requiere más comparaciones que la cota dada por la teoría de la información cuando se ordena ese número de elementos.

Problema 12.8. Prosiguiendo con el problema 12.7, hallar una fórmula explícita para el número de comparaciones que se efectúa en el caso peor empleando *ordenar-fusión* en el caso general, como función del número n de elementos que hay que ordenar. Buscar el menor entero postivo tal que *ordenar-fusión* requiere más comparaciones que la cota dada por la teoría de la información cuando se ordena ese número de elementos.

Problema 12.9. Supongamos que pedimos a nuestro algoritmo de ordenación no solo que determine el orden de los elementos, sino también que determine cuales son iguales, si los hubiera. Por ejemplo, no es aceptable un veredicto como $A < B \leq C$: el algoritmo debe especificar si $B = C$ *ó* si $B < C$. Dar una cota inferior de teoría de la información para el número de comparaciones que se requiere en el caso peor para manejar n elementos. Rehaga este problema si para cada comparación hay tres posibles resultados: "<", "=" y ">".

Problema 12.10. Sea $T[1..n]$ un vector, y sea $k \leq n$ un entero. El problema consiste en devolver, por orden descendente, los k elementos mayores de T. Demostrar mediante un argumento de teoría de la información que todo algoritmo determinista basado en comparaciones que resuelva este problema tendrá que efectuar al menos $\frac{k}{2} \lg \frac{n}{2}$ comparaciones, tanto en el caso peor como en el caso promedio. Concluir que esto tiene que requerir un tiempo que esté en $\Omega(k \log n)$. Por otra parte, dar un algoritmo capaz de resolver este problema en un tiempo que esté en $O(n \log k)$ y que utilice un espacio que esté en $O(k)$ en el caso peor. El algoritmo no debe hacer más de una pasada secuencial por el vector T. Justifique su análisis del tiempo y el espacio empleados por el algoritmo.

Problema 12.11. Dar un árbol de decisión completo para el problema de las doce monedas de la Sección 12.2.2. Cada nodo del árbol debe especificar qué monedas están en cada platillo de la balanza. El hijo izquierdo de los nodos internos da la próxima medida que hay que hacer si el fiel se inclina a la izquierda, e igualmente el hijo medio y el hijo derecho si el fiel queda equilibrado o se inclina a la derecha, respectivamente. Omita los descendientes del lado derecho de la raíz, para hacer que el árbol sea más pequeño; estos descendientes se pueden manejar por simetría. Llamar a las monedas de la forma $A, B, C, \ldots, L$, para que la raíz del árbol de decisión sea *ABCD:EFGH* de acuerdo con la figura 12.6 y con el razonamiento de teoría de la información dado en la Sección 12.2.2.

Problema 12.12. Prosiguiendo con el problema 12.11, demostrar mediante un argumento de teoría de la información que el problema de las monedas y la balanza no se puede resolver utilizando solo tres veces la balanza si tenemos 13 monedas en lugar de 12. Este resultado es interesante porque a primera vista parece posible, que este problema se pudiera resolver: 13 monedas generan 27 veredictos potenciales, y un árbol ternario de altura 3 tiene 27 hojas para dar cabida a todos ellos.

Problema 12.13. Prosiguiendo con los problemas 12.11 y 12.12, demostrar que el problema de *cuatro* monedas no se puede resolver con solo dos medidas, a no ser que se disponga de una moneda adicional de peso "correcto".

Problema 12.14. Utilice un argumento del adversario para demostrar que todo algoritmo basado en comparaciones para decidir si un valor buscado aparece entre n posibilidades tiene que requerir un tiempo que esté en $\Omega(\log n)$ en el caso peor, independientemente de la habilidad con la que se disponga la tabla de posibilidades. En particular, esta cota inferior se alcanza mediante una búsqueda binaria si la tabla está ordenada. ¿Sabría demostrar esto mediante un argumento de teoría de la información? ¿Hasta qué punto es importante limitar nuestra atención a los algoritmos basados en comparaciones?

Problema 12.15. Sea $T[1..n]$ un vector ordenado formado por enteros diferentes, algunos de los cuales pueden ser negativos. El problema 7.12 pedía un algoritmo para hallar un índice i tal que $1 \leq i \leq n$ y $T[i] = i$, siempre y cuando exista ese índice, en un tiempo que estuviera en $O(\log n)$ en el caso peor. Utilice un argumento de teoría de la información para demostrar que todo algoritmo basado en comparaciones que resuelva este problema tiene que requerir un tiempo que esté en $\Omega(\log n)$. Por otra parte, demostrar mediante un argumento del adversario que todo algoritmo basado en comparaciones que resuelva este problema requeriría un tiempo en $\Omega(n)$si no fuera por la restricción de que los elementos de T sean distintos.

Problema 12.16. Utilizar un argumento del adversario para demostrar que todo algoritmo determinista correcto para decidir si un grafo es conexo tiene que investigar todos los pares $\{i, j\}$ de vértices tanto si existe una arista entre i y j como si no. Suponga, tal como en la Sección 12.3.2, que las únicas preguntas que se admiten acerca del grafo son del tipo "¿Existe una arista entre los vértices i y j?"

Problema 12.17. Considere el problema consistente en determinar si un grafo no dirigido con n nodos contiene un camino cuya longitud es 2 como mínimo.

(a) Utilice un argumento del adversario para demostrar que cualquier algoritmo para resolver este problema debe requerir un tiempo que esté en $\Omega(n^2)$ en el caso peor si se limita a hacer preguntas de la forma "¿Existe una arista entre los nodos i y j?" (Tal sucede si el grafo se representa mediante una matriz de adyacencia: **tipo** *grafoadya* en la Sección 5.4).

(b) Demostrar, por otra parte, que este problema se puede resolver en un tiempo que está en $O(n)$ si el algoritmo puede pedir a cada vértice la lista de nodos adyacentes. (Tal sucede si el grafo se representa mediante listas de adyacencia: **tipo** *grafolista* en la Sección 5.4).

Problema 12.18. Ya se vio que todo algoritmo correcto basado en comparaciones para hallar la mediana de n elementos tiene que efectuar al menos $3(n-1)/2$ comparaciones en el caso peor. Utilice un argumento del adversario mucho más sencillo para demostrar que cuando todos los elementos son distintos no es posible localizar la mediana con certeza sin examinarlos todos. Por otra parte, demostrar mediante un ejemplo que esto no es cierto si los elementos no son distintos.

Problema 12.19. El algoritmo evidente para hallar tanto el mínimo como el máximo de los elementos de una matriz de n elementos requiere 2n–3 comparaciones. Demostrar

mediante un argumento antagónico que todo algoritmo basado en comparaciones para este problema requiere al menos $\lceil 3n/2 \rceil - 2$ comparaciones en el caso peor. (Opcionalmente: buscar un algoritmo que satisfaga esta cota inferior).

Problema 12.20. Mostrar la forma de emplear un algoritmo de factorización para descomponer números compuestos, y para decidir acerca de la primalidad de números arbitrarios. (¡No queremos decir que la mejor manera de comprobar la primalidad sea la factorización!)

Problema 12.21. Suponga que se representan enteros muy grandes en un vector en base decimal. Por ejemplo, $T[1..n]$ representa el número entero $\sum_{i=1}^{n} 10^{i-1} T[i]$. Dar un algoritmo para efectuar la división por 4 de tales enteros que requiera un tiempo en $O(n)$. Analizar el tiempo requerido por el algoritmo. Generalizarlo para bases distintas de 10.

Problema 12.22. Demostrar que es imposible que una función que esté en $\Omega(2^n)$ sea suave.

Problema 12.23. Demostrar que toda función fuertemente cuadrática es al menos cuadrática; véase la Sección 12.4.1.

Problema 12.24. Prosiguiendo con el problema 12.23, dar un ejemplo explícito que muestre que era necesario especificar en la definición de funciones fuertemente cuadráticas que tienen que ser eventualmente no decrecientes. Específicamente, mostrar una función $t: \mathbb{N} \to \mathbb{R}^{\geq 0}$ tal que $t(an) \geq a^2 t(n)$ para todo entero positivo a y todo entero n suficientemente grande, y tal que sin embargo $t(n)$ no sea al menos cuadrática.

Problema 12.25. Dar un ejemplo explícito de una función asintóticamente no decreciente que sea al menos cuadrática pero no fuertemente cuadrática.

Problema 12.26. Una función $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ es *supra cuadrática* si es asintóticamente no decreciente y existe un ε positivo tal que $t(an) \geq a^{2+\varepsilon} t(n)$ para todo entero positivo a y para todo entero n suficientemente grande. Demostrar que $n^2 \log n$ es fuertemente cuadrática pero no supra cuadrática.

Problema 12.27. Sean **SQR**, **MLT** y **DIV** los problemas consistentes en elevar al cuadrado un número entero de tamaño n, multiplicar dos enteros de tamaño n y determinar el cociente cuando se divide un entero de tamaño $2n$ por un entero de tamaño n, respectivamente. Claramente, estos problemas son al menos lineales porque todo algoritmo que los resuelva tiene que tener en cuenta todos y cada uno de los bits de los operandos implicados. Suponiendo que los tres problemas sean suaves y que **MLT** sea fuertemente lineal, demostrar que los tres problemas son linealmente equivalentes.

Sugerencia: Si $10^{n-1} \leq i \leq 10^n - 1$, se define el *pseudoinverso* i^* como $10^{2n-1} \div i$. Por ejemplo, $36^* = 27$ y $27^* = 37$. (En la práctica, es posible que no utilizásemos la base 10). Sea **INV** el problema consistente en calcular el pseudoinverso de un entero de tamaño n.

Utilice **INV** en su cadena de reducciones. Por ejemplo, demuestre que $i \div j \approx (i \times j^*) \div 10^{2n-1}$ cuando i y j son enteros de tamaño (decimal) $2n$ y n, respectivamente, y muestre la forma de utilizar esto para concluir que $\mathbf{DIV} \leq^{\ell} \mathbf{INV}$ se sigue de $\mathbf{MLT} \leq^{\ell} \mathbf{SQR} \leq^{\ell}$.

Problema 12.28. Prosiguiendo con el problema 12.27, sea **SQRT** el problema consistente en hallar la parte entera de la raíz cuadrada de un entero de tamaño n. Demostrar, haciendo suposiciones razonables, que

SQRT y **MLT** son linealmente equivalentes. Enuncie explícitamente las suposiciones que necesite.

Problema 12.29. Denotar por **SU** el problema consistente en elevar al cuadrado una matriz triangular cuya diagonal consta únicamente de unos. Demuestre que **SU** ≡ **MQ** haciendo suposiciones razonables. ¿Qué suposiciones necesita?

Problema 12.30. Probar que la inversa de una matriz no singular triangular superior es triangular superior.

Problema 12.31. Denote por **IQ** el problema consistente en invertir una matriz arbitraria no singular. Demuestre que **IQ** ≡ **MQ**, suponiendo que tanto **IQ** como **MQ** son supra cuadráticos; véase el problema 12.26. Observe que esta reducción *no* sería posible si existiera un algoritmo que fuera capaz de multiplicar matrices $n \times n$ en un tiempo que estuviera en $O(n^2 \log n)$, porque esta función no es supra cuadrática.

Problema 12.32. Demostrar que **MUL** $\leq^{\ell}$ **TCR**, suponiendo que **TRC** sea suave.

Problema 12.33. Demostrar que **TCR** $\leq^{\ell}$ **MUL**, suponiendo que **MUL** sea suave y fuertemente cuadrático.

Problema 12.34. Mostrar un árbol de decisión para el cual sea posible demostrar que a todo algoritmo que lo resuelva le corresponde otro algoritmo que lo resuelve exponencialmente más deprisa en todos los casos salvo un número finito.

Problema 12.35. Demostrar que se necesita un tiempo polinómico para ejecutar un número polinómico de operaciones de tiempo polinómico.

Problema 12.36. El conjunto de todos los números compuestos está evidentemente en NP, puesto que todo divisor no trivial de n es una evidencia convincente de que n es compuesto. Si recordamos la asimetría de NP mencionada inmediatamente antes del Teorema 12.5.3, resulta tentador esperar que el conjunto complementario, formado por todos los números primos, no esté en NP. Después de todo, a primera vista parecerá que no hay una forma sucinta de demostrar que un número es primo: ¿qué podríamos mostrar para demostrar la *in*existencia de un divisor no trivial? Sin embargo, en este mundo no hay nada seguro, salvo la muerte y los impuestos: también la primalidad está en NP, aun cuando la noción de certificado de primalidad es bastante más sutil que la noción de certificado de no primalidad. ¡Ese es el problema que le planteamos!

Pista: Un entero impar $n > 2$ es primo si y solo si existe un entero x entre 1 y n–1 tal que $x^{n-1} \bmod n = 1$ y $x^{(n-1)/p} \bmod n \neq 1$ para todo divisor primo p de n–1.

Problema 12.37. Demostrar que las reducciones polinómicas son transitivas. Considere unos problemas **A**, **B** y **C** cualesquiera, tales que $\mathbf{A} \leq_T^p \mathbf{B}$ y $\mathbf{B} \leq_T^p \mathbf{C}$; demuestre que $\mathbf{A} \leq_T^p \mathbf{C}$. Rehaga el problema con reducciones muchos a uno "$\leq_m^p$", con la restricción de que **A**, **B** y **C** son problemas de decisión.

Problema 12.38. Mostrar dos problemas de decisión muy sencillos X e Y tales que $X \leq_T^p Y$ pero $X \not\leq_m^p Y$.

Problema 12.39. Mostrar dos problemas de decisión X e Y tales que $X \leq_T^p Y$, y sin embargo hay razones fundadas para creer que $X \not\leq_m^p Y$. Para hacer más interesante este problema que el problema 12.38, los conjuntos X e Y deben ser infinitos, y también deben serlo sus complementos.

Problema 12.40. Prosiguiendo el problema 12.39, mostrar dos problemas de decisión X e Y tales que $X \leq_T^p Y$, y sin embargo sea posible demostrar que $X \not\leq_m^p Y$. Una vez más, X e Y deben de ser infinitos y también deben de serlo sus complementos.

Problema 12.41. Considere dos problemas de decisión X e Y. Demuestre que si $X \in \text{NP}$ y $Y \leq_m^p X$, entonces $Y \in \text{NP}$.

Problema 12.42. Prosiguiendo con el problema 12.41, dar una evidencia convincente de que es posible que $X \in \text{NP}$ e $Y \leq_T^p X$, y sin embargo $Y \notin \text{NP}$ aun cuando Y sea un problema de decisión.

Problema 12.43. Demostrar que el problema del coloreado óptimo de un grafo es reducible a decisión. Específicamente, considere los problemas **COLD**, **COLO** y **COLC** especificados en la Definición 12.5.8. Demuestre que estos tres problemas son polinómicamente Turing equivalentes.

Problema 12.44. Prosiguiendo con el problema 12.43, demuestre que el problema 3COL, que también se especificaba en la Definición 12.5.18, es polinómicamente Turing equivalente a **COLD**, **COLO** y **COLC**. Puede dar por bueno el resultado requerido en el problema 12.43, y puede utilizar el hecho de que **3COL** es NP-completo.

Problema 12.45. Dado un grafo no dirigido $G = \langle N, A \rangle$, una *camarilla* es un conjunto de nodos tal que existe una arista del grafo entre dos nodos cualesquiera de la camarilla. (En ocasiones, se define la camarilla como un conjunto máximo que tenga esta propiedad; nosotros no hacemos hincapié en esta condición). Hay tres problemas naturales que conciernen a las camarillas:

◊ **CLQD:** Dado un grafo G y un entero k, ¿existe una camarilla de tamaño k en G?
◊ **CLKO:** Dado un grafo G, hallar el tamaño de la mayor camarilla que exista en G.
◊ **CLKC:** Dado un grafo G, hallar una camarilla de tamaño máximo en G.

Demuestre que el problema de la camarilla es reducible a decisión: los tres problemas anteriores son polinómicamente Turing equivalentes.

Problema 12.46. Demostrar que el problema de hallar una asignación que satisfaga en una fórmula booleana es reducible a decisión: el problema de hallar una asignación que satisfaga para una fórmula booleana es polinómicamente reducible al problema de decidir si existe una de estas asignaciones.

Problema 12.47. Demostrar que el problema del viajante es reducible a decisión: **TSP** $\equiv_T^p$ **TSPD**. (Estos problemas se definen inmediatamente antes del Teorema 12.5.9).

Problema 12.48. Considere el problema de decisión

$F = \{\langle n, d \rangle \mid n \in \mathbb{N}^+ \text{ y } n \text{ tiene un divisor no trivial menor que } d\}$

Demostrar que F es polinómicamente Turing equivalente al problema de factorización.

Problema 12.49. Demostrar que cualquier par de problemas NP-completos es polinómicamente equivalente Turing.

Problema 12.50. Demostrar que **HAMD**, el problema de decidir si un grafo es hamiltoniano, es NP-completo.

Problema 12.51. Demostrar que **COLD** es NP-completo; véase la Definición 12.5.18.

Problema 12.52. Demostrar que **3COL** es NP-completo aun cuando nos limitemos a grafos planares de grado no mayor que 4; véase la Definición 12.5.8.

Problema 12.53. Buscar un algoritmo de tiempo polinómico para resolver **SAT-2-CNF**.

Problema 12.54. Considere el problema consistente en decidir la viabilidad de las fórmulas booleanas en **CNF** tales que cada cláusula contenga *exactamente* tres literales. Demuestre que este problema es NP-completo, reduciendo a él el problema **SAT-3-CNF**.

Problema 12.55. Demostrar que **CLQD**, la versión de decisión del problema de la camarilla definido en el problema 12.45, es NP-completo.

Problema 12.56. Se nos da una colección $x_1, x_2, \ldots x_n$ formada por n números enteros. Nuestra tarea consiste en decidir si existe o no un conjunto $X \subseteq \{1, 2, \ldots, n\}$ tal que $\sum_{i \in X} x_i = \sum_{i \notin X} x_i$. Esto se conoce con el nombre de PARTICIÓN.

(a) Demostrar que este problema es NP-completo.
(b) Demostrar que es reducible a decisión: un oráculo para resolver el problema de decisión se puede utilizar en tiempo polinómico para hallar un X adecuado siempre exista.

Problema 12.57. Demostrar que si $X \in$ NP e Y es de NP-difícil, entonces $X \leq_T^p Y$. En otras palabras, que los problemas NP-difíciles son al menos tan difíciles como cualquier problema que esté en NP.

Problema 12.58. Dar explícitamente un algoritmo no determinista que resuelva el problema de no primalidad en un tiempo polinómico. Analizar el tiempo de ejecución del algoritmo.

Problema 12.59. Prosiguiendo con los problemas 12.36 y 12.58, dar explícitamente un algoritmo no determinista que resuelva en tiempo polinómico el problema de primalidad. Analizar el tiempo de ejecución del algoritmo.

Problema 12.60. Demuestre que NP⊆NPSPACE. Para hacerlo, observe que basta una cantidad polinómica de espacio para enumerar todos los posibles certificados acotados polinómicamente e ir probándolos, para ver si al menos uno resulta aplicable.

Problema 12.61. Demostrar que PSPACE = NPSPACE. Para hacerlo, mostrar que si $s(n) \geq \lg n$ se puede calcular eficientemente, entonces todo problema de decisión que se pueda resolver empleando una cantidad de espacio en bits que esté en $O(s(n))$ mediante un algoritmo no determinista, se puede resolver también mediante una cantidad de espacio en bits que esté en $O(s(n^2))$ mediante un algoritmo determinista.

Problema 12.62. Demostrar que LOGSPACE ⊆ P. Para hacerlo, observe que todo algoritmo determinista que se encuentre dos veces en la misma configuración cae en un bucle infinito; solo hay 2^s configuraciones diferentes cuando se dispone de s bits de almacenamiento; y $2^{k \log n} = n^k$.

Problema 12.63. Suponiendo que NP ≠ co-NP, demostrar que los problemas NP-completos no pueden pertenecer a co-NP.

Problema 12.64. Demostrar que RP ⊆ NP. Para ello, observamos que toda sucesión de elecciones probabilistas que lleve a un algoritmo probabilista RP a una admisión es una evidencia convincente de que el caso considerado es un caso-si.

Problema 12.65. Demostrar que ZPP=RP∩co-RP. Observe el parecido con el problema 10.28.

Problema 12.66. Dos grafos $G = \langle V, A \rangle$ son *isomorfos* si existe una correspondencia entre los vértices de G y los de H que mantenga la adyacencia. Formalmente, G y H son isomorfos si existe una función biyectiva $s:V \text{ Æ } W$ tal que $(v_1, v_2) \in A$ si y solo si $(\sigma(v_1), \sigma(v_2) \in B$ para todo $v_1, v_2 \in V$. Aun cuando no se conoce ningún algoritmo de tiempo polinómico para decidir si dos grafos son o no isomorfos, este problema está evidentemente en NP puesto que la función s podría servir como certificado. No obstante, se cree que el problema de *no isomorfismo* de grafos no está en NP: ¿qué tipo de evidencia sucinta podría demostrar que dos grafos no son isomorfos? Sin embargo, el problema planteado es mostrar que el no isomorfismo de grafos pertenece a la clase IP.

Sugerencia: Si de hecho G y H son isomorfos, y si me dan un grafo K seleccionado aleatoriamente entre todos los grafos isomorfos a G, no hay forma de saber si K se ha producido a partir de G o a partir de H.

12.8 REFERENCIAS Y TEXTOS MÁS AVANZADOS

La complejidad computacional se abarca detalladamente en *Papadimitriou* (1994). En *Pippenger* (1978) se da una introducción para quienes no sean especialistas. La demostración de *Wells* de que se necesitan 30 comparaciones en el caso peor para ordenar 12 elementos es un ejemplo de cálculo masivo: se necesitaron 60 horas de una (entonces puntera) computadora MANIAC II. Véase *Gonnet* y *Munro* (1986) y *Carlsson* (1987a) para hallar unas modificaciones de *ordenar-mont* que lo hacen aproximarse mucho a resultar óptimo a efectos del número de comparaciones en el caso peor. Es posible ordenar en un tiempo más rápido que $\Theta(n \log n)$ empleando una técnica de fusión de árboles presentada por *Fredman* y *Willard* (1990) aun cuando los elementos que hay que ordenar son demasiado grandes para emplear la ordenación por casillero; por supuesto, esta técnica de ordenación no se basa en comparaciones.

La noción de problemas suaves y su aplicación a las reducciones lineales se originó en *Brassard* y *Bratley* (1988). La reducción lineal de la división entera a la multiplicación entera se debe a *Cook* y *Aanderaa* (1969). Para más información acerca de las reducciones entre problemas aritméticos, consulte *Aho, Hopcroft* y *Ullman* (1974). La reducción de la inversión de matrices arbitrarias no singulares a la multiplicación de matrices procede de *Bunch* y *Hopcroft* (1974). Si f y g son funciones de coste, como en la Sección 12.4.3, en *Fredman* (1976) se da un algoritmo asintóticamente más rápido que el algoritmo evidente para calcular fg. La reducción lineal de la multiplicación de funciones de coste al cálculo de cierres reflexivos transitivos se debe a *Fischer* y *Meyer* (1971) y la reducción inversa es de *Furman* (1970); entre la dos demuestran el Teorema 12.4.12. En el caso de las funciones de coste cuyo intervalo está limitado a $\{0, +\infty\}$, *Arlazarov, Dinic, Kronrod* y *Faradžev* (1970) presentan un algoritmo para calcular fg empleando un número de operaciones *booleanas* que está en $\Theta(n^3 / \log n)$; el Teorema 12.4.13 es de *Fischer* y *Meyer* (1971).

La teoría de la NP completitud se originó en dos artículos fundamentales: *Cook* (1971) demuestra que **SAT-CNF** es NP-completo, y *Karp* (1972) subraya la importancia de esta noción, presentando un gran número de problemas NP-completos. Para ser históricamente

exactos, la afirmación original de Cook es que $X \leq_T^p$ **TAUT – DNF** para todo $X \in$ NP, donde **TAUT-DNF** concierne a las tautologías en forma normal disyuntiva; sin embargo, es probable que este problema no sea NP-completo porque no pertenece a NP a no ser que NP = co-NP. *Levin* (1973) desarrolló independientemente una teoría similar, empleando problemas de teselado en lugar de tautologías. La idea de que el tiempo polinómico es un concepto fundamental era ya conocida por *Cobham* (1964) y *Edmonds* (1965). La autoridad indiscutible en temas de NP-completitud es *Garey* y *Johnson* (1979). También proporciona una buena introducción la obra de *Hopcroft* y *Ullman* (1979).

El término "reducible a decisión" les fue sugerido a los autores por Papadimitriou; lea *Bellare* y *Goldwasser* (1994) para más información acerca de la complejidad de decisión frente a la de búsqueda. La reducibilidad a decisión no debe confundirse con la noción, más conocida, de autorreducibilidad, según la cual las soluciones de muchos problemas se pueden reducir a resolver el mismo problema en casos más pequeños. Véase *Naik, Ogiwara* y *Selman* (1993) para más detalles acerca de la forma en que están relacionadas la reducibilidad a decisión y la autoreducibilidad.

Las clases de complejidad probabilistas fueron investigadas por *Gill* (1977); la clase RP se debe a *Adleman* y *Manders* (1977). Las demostraciones interactivas y la clase IP proceden de *Goldwasser, Micali* y *Rackoff* (1989); una idea parecida fue desarrollada independientemente por *Babai* y *Moran* (1988). La primera investigación rigurosa de NC se debe a *Pippenger* (1979). La computación cuántica se originó en *Benioff* (1982), *Feynman* (1982, 1986) y *Deutsch* (1985); véase también *Deutsch* y *Jozsa* (1992), *Bernstein* y *Vazirani* (1993), *Lloyd* (1993), *Berthiaume* y *Brassard* (1994), *Shor* (1994) y *Simon* (1994). Una fuente enciclopédica de información sobre el zoo de las clase de complejidad clásicas es la obra de *Johnson* (1990); véase también *Papadimitriou* (1994).

Muchos de los problemas que conciernen a reducciones polinómicas en la Sección 12.7 están resueltos en *Karp* (1972). El problema 12.34 es de *Blum* (1967). El hecho de que el conjunto de los números primos está en NP (problemas 12.36 y 21.59) es de *Pratt* (1975); se dan más certificados sucintos de primalidad en *Pomerance* (1987). Los problemas 12.48 y 12.63 proceden de *Brassard* (1979). Parte de la solución del problema 12.52 es de *Stockmeyer* (1973). El problema 12.61 es de *Savitch* (1970). El problema 12.66 procede de *Goldreich, Micali* y *Widgerson* (1991).

Hay varias técnicas importantes de complejidad computacional que no se han mencionado en este capítulo. En *Aho, Hopcroft* y *Ullman* (1974), *Borodin* y *Munro* (1975) y *Winograd* (1980) se describe una aproximación algebraica a las cotas inferiores. Aunque no sabemos demostrar que no haya algoritmos eficientes para los problemas NP-completos, existen problemas que son intrínsecamente difíciles, según se describe en *Aho, Hopcroft* y *Ullman* (1974). Estos se pueden resolver en teoría, pero se puede demostrar que ningún algoritmo puede resolverlos en la práctica cuando los casos son de tamaño moderado, ni siquiera admitiendo que se requiera un tiempo comparable con la edad del Universo y tantos bits de almacenamiento como partículas elementales hay en el Universo conocido; véase *Stockmeyer* y *Chandra* (1979). También existen problemas que no se pueden resolver mediante algoritmo alguno, sean cuales fuesen los recursos disponibles; léase *Turing* (1936), *Gardner* y *Bennet* (1979), y *Hopcroft* y *Ullman* (1979) para una discusión acerca de estos problemas *indecidibles*.

Capítulo 13

Algoritmos heurísticos y aproximados

El conocimiento de que un problema es difícil de resolver es instructivo. Puede incluso ser útil si evita que desperdiciemos tiempo buscando un algoritmo que probablemente no exista. Sin embargo, esto no hace que el problema desaparezca. En algunas ocasiones, uno tiene que hallar algún tipo de solución para el problema, tanto si es difícil como si no. Éstos son los dominios de la heurística y de los algoritmos aproximados.

Al hablar de *algoritmos heurísticos*, simplemente *heurística*, nos referimos a un procedimiento que puede producir una buena solución para nuestro problema, incluso una solución óptima si somos afortunados, pero que por otra parte puede no producir una solución, o dar lugar a una que no sea precisamente óptima, si no lo somos. La heurística puede ser probabilística o determinista. La diferencia esencial entre una heurística probabilística y un algoritmo de Monte Carlo es que este último tiene que buscar una solución correcta con probabilidad positiva (preferiblemente alta), sea cual fuere el caso considerado. Por otra parte, puede haber casos para los cuales la heurística, tanto si es probabilista como si no, nunca producirá una solución. En algunos casos, un procedimiento heurístico puede ser poco más que una estimación inteligente. Un ejemplo ya visto es la «heurística minimax» descrita en la Sección 9.8.

Reservaremos el término *algoritmo aproximado* para un procedimiento que siempre proporcione algún tipo de solución para el problema, aun cuando quizá no llegue a encontrar la solución óptima. Para que sea útil, también debe ser posible calcular una cota bien de la diferencia, bien de la razón entre la solución óptima y la producida por el algoritmo aproximado. Las secciones siguientes ilustran distintas clases de cotas que pueden hallarse.

Aun cuando esté disponible un algoritmo exacto, a veces puede ser útil un buen algoritmo aproximado. Los datos involucrados en un cierto caso que haya que resolver son frecuentemente inciertos o imprecisos hasta cierto punto: la dificultad de recoger información en las situaciones prácticas es tal que raramente resulta posible ser preciso. Si el error causado por el algoritmo aproximado es me-

nor que el posible error causado por el uso de datos inexactos, y si el algoritmo de aproximación es más eficiente que el algoritmo exacto, entonces a todos los efectos prácticos puede ser preferible utilizar el primero.

13.1 ALGORITMOS HEURÍSTICOS

13.1.1 Coloreado de un grafo

Nos enfrentamos por primera vez con los problemas de coloreado en la Sección 12.5.4. En la definición 12.5.18 se definen formalmente varios problemas asociados. El problema concreto que nos concierne en esta sección es el siguiente.

Sea $G = <N, A>$ un grafo no dirigido. Deseamos colorear los nodos de G de tal manera que no haya dos nodos adyacentes del mismo color. El problema pregunta cual es el mínimo número de colores que se necesita. Este mínimo se denomina *número cromático* del grafo. Por ejemplo, el grafo de la figura 13.1 se puede colorear empleando sólo dos colores, digamos el rojo para los nodos 1, 3 y 4 y el azul para los nodos 2 y 5. Tal como se vio en la Sección 12.5.5, este problema es de dificultad NP.

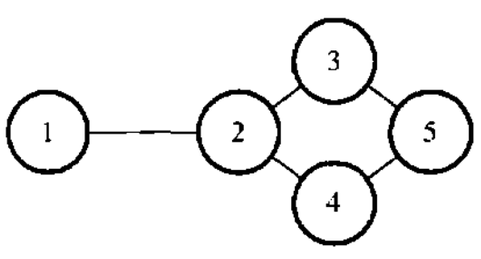


Figura 13.1. Un grafo para colorear

La heurística voraz evidente consiste en seleccionar un color y un nodo inicial arbitrario, y considerar después cada nodo por turnos. Si se puede pintar un nodo con el primer color (en otras palabras, si todavía no se ha pintado ninguno de sus vecinos), lo hacemos así. Cuando ya no se pueden pintar más nodos, seleccionamos un nuevo color y un nuevo nodo inicial que todavía no esté pintado. Y entonces pintamos todos los nodos que podamos con este segundo color; no importa si uno o más de sus vecinos están pintados ya con el primer color. Si todavía quedan nodos sin pintar, seleccionamos un tercer color, pintamos con él todos los nodos que podamos, y así sucesivamente.

En el ejemplo, si el nodo 1 se pinta de rojo, entonces no podemos pintar el nodo 2 del mismo color. Sin embargo, los nodos 3 y 4 se pueden pintar los dos de rojo, pero entonces el nodo 5 no se puede pintar. Si comenzamos de nuevo en el nodo 2 empleando pintura azul, entonces podemos pintar los nodos 2 y 5 y terminar nuestra tarea empleando tan sólo dos colores. Se trata de una solución óptima para nuestro ejemplo, puesto que evidentemente no es posible una solución que utilice nada más que un color. Suponga, sin embargo, que consideramos sistemáticamente los nodos por el orden 1, 5, 2, 3, 4. Ahora obtenemos una respuesta distin-

ta: en este caso los nodos 1 y 5 se pueden pintar de rojo, y entonces el nodo 2 se puede pintar de azul, pero los nodos 3 y 4 requieren un tercer color, puesto que ya tienen ambos un vecino rojo y otro azul. En este caso, el resultado no es óptimo. El algoritmo voraz, por tanto, no es más que una heurística que posiblemente pueda hallar una solución óptima, pero sin certeza de que así sea.

Aun cuando la heurística puede no hallar una solución óptima, podemos tener la esperanza de que en la práctica pueda encontrar una solución «buena», no muy distinta de la óptima. Veamos si nuestra esperanza está justificada.

En primer lugar, no es difícil mostrar que para todo grafo G existe al menos una ordenación de los nodos que permite al algoritmo hallar una solución óptima. En otras palabras, sea cual fuere el grafo en que estemos trabajando, siempre existe la posibilidad de tener suerte y hallar una solución óptima. Para ver esto, considere un grafo cualquiera G y suponga que una solución óptima requiere k colores. Suponga además que por arte de magia le dan una forma de colorear G empleando tan solo k colores. Numere arbitrariamente estos k colores, y numere después los nodos de G de la forma siguiente: primero se numeran consecutivamente todos los nodos de G que se hayan pintado con el color 1 en la solución óptima. A continuación sigue la sucesión numerando todos los nodos que sea hayan pintado con el color 2 en la solución óptima, y así sucesivamente. Cuando termine con el color k, todos los nodos estarán numerados. Entre nodos del mismo color, no importa cuál se numere primero.

Si ahora se aplica la heurística voraz al grafo G considerando los nodos por el mismo orden que los números que acabamos de asignar, está claro que encontrará una solución óptima. Sin embargo, puede que no sea la misma que nos dieron como punto de partida. En efecto, suponga que aplica el color 1. Ciertamente, podrá pintar todos los nodos que tuvieran el color 1 en la solución original, y quizá pueda pintar algunos más. Cuando aplique el color 2, algunos nodos que tenían este color en la solución original pueden estar ya pintados del color 1. Sin embargo, ciertamente habrá uno o más nodos que tuvieran el color 2 en la solución original y que no hayan sido pintados (el problema 13.2 pide justificar esta afirmación). Será posible pintarlos todos del color 2, y quizá algunos otros nodos (la presencia de nodos adicionales del color 1 no puede hacer imposible pintar del color 2 un nodo que no esté pintado). Prosiguiendo de esta manera, cuando acabemos el color 1 habremos pintado al menos tantos nodos como tuvieran el color 1 en la solución original; cuando acabemos el color 2, habremos pintado al menos tantos nodos como tuvieran el color 1 o el 2 en la solución original; y así sucesivamente, hasta que al acabar el color k hayamos pintado al menos tantos nodos como se hubieran pintado de los colores 1, 2,..., k en la solución original: en otras palabras, se habrán pintado todos los nodos. Entonces habremos hallado una solución que utiliza tan sólo k colores, lo cual es óptimo.

En el lado negativo, hay grafos que hacen que esta heurística sea tan mala como queramos. Más exactamente, hay grafos que se pueden colorear con sólo k colores, y para los cuales, si no somos afortunados, la heurística encontrará una solución empleando c colores con c/k tan grande como deseemos. Para ver esto, considere un grafo que tiene $2n$ nodos, numerados de 1 a $2n$. Cuando i es impar, el no-

do 1 es adyacente a todos los nodos de número par salvo el nodo i+1; cuando i es par, el nodo i es adyacente a todos los nodos de número impar salvo el nodo $i-1$. La figura 13.2 muestra este grafo para el caso en que n=4. Este grafo es *bipartito*: los nodos se pueden dividir en dos conjuntos N_1 y N_2 (los nodos de número par e impar respectivamente, en este ejemplo) de tal manera que cada arista une un nodo del conjunto N_1 con un nodo del conjunto N_2. Estos grafos siempre se pueden colorear con sólo dos colores.

Por ejemplo, podríamos pintar de rojo los nodos impares, y de azul los nodos pares. La heurística voraz hallará esta solución óptima si intenta pintar los nodos por el orden 1, 2,..., $2n-1$, 2, 4,..., $2n$, por ejemplo. Por otra parte, si examina los nodos por su orden natural 1, 2,...,$2n-1$, $2n$, entonces es fácil ver que hallará una solución que requiere n colores: los nodos 1 y 2 se pueden pintar del color 1, los nodos 3 y 4 deben de pintarse con un nuevo color 2, los nodos 5 y 6 deben de pintarse con un nuevo color 3, y así sucesivamente. Seleccionando un n suficientemente grande, podemos hacer que esta solución sea tan mala como nos parezca oportuno.

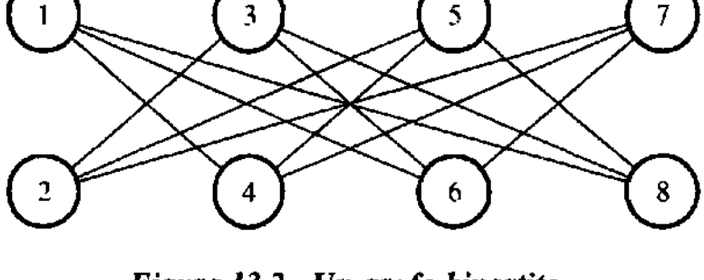


Figura 13.2. Un grafo bipartito

Este procedimiento es, decididamente, heurístico y no un algoritmo aproximado: no hay forma de establecer una cota del error en la solución proporcionada. Podríamos intentar protegernos contra los errores más graves ejecutando varias veces la heurística aplicada al mismo grafo, con distintas ordenaciones de las aristas seleccionadas al azar. Aun así, la ausencia de una cota verdadera del error hace que el procedimiento tenga escaso valor práctico. Para un grafo con n nodos, en el caso peor (cuando el grafo sea completo) la heurística podría tener que utilizar n colores; para cada color, tiene que examinar todos y cada uno de los n nodos; y para cada nodo que no esté pintado todavía podría tener que examinar $n-1$ nodos vecinos. El tiempo de ejecución, por tanto, está en $O(n^3)$.

13.1.2 El viajante

Este problema se presentó en la Sección 12.5.2. Conocemos las distancias entre un cierto número de ciudades. El viajante desea salir de una de estas ciudades, para visitar todas las demás ciudades exactamente una vez y volver al punto de partida habiendo recorrido la menor distancia posible. Suponemos que la distancia entre dos ciudades nunca es negativa. Al igual que el problema anterior, el problema del viajante es de dificultad NP, y los algoritmos que se conocen no resultan prácticos para ejemplares muy grandes. (Por supuesto, esto depende de lo que se quiera decir con «grande». Empleando programación lineal y técnicas de corte de pla-

nos de un tipo que no se ha descrito en este libro, se pueden resolver en la actualidad problemas que impliquen algunos centenares de ciudades. En el momento de escribir este libro, el mayor caso real del problema que se había resuelto constaba de 4.461 ciudades; véase la Sección 11.9).

El problema se puede representar como un grafo completo no dirigido con n nodos. (El grafo también puede ser dirigido si la matriz de distancias no es simétrica. Aquí no consideraremos esta posibilidad.) Recordamos al lector que un ciclo del grafo que pase por todos los nodos exactamente una vez se denomina *ciclo hamiltoniano*. Nuestro problema, por tanto, requiere que hallemos el ciclo hamiltoniano más corto de un grafo dado.

Por ejemplo, supongamos que nuestro problema concierne a seis ciudades con la siguiente matriz de distancias.

Desde:	Hacia 2	3	4	5	6
1	3	10	11	7	25
2		8	12	9	26
3			9	4	20
4				5	15
5					18

En este caso, el recorrido óptimo tiene una longitud de 58. Esta longitud se puede lograr en un recorrido que visite los nodos 1, 2, 3, 6, 4 y 5 por ese orden, antes de volver al punto de partida en el nodo 1.

La heurística voraz evidente consiste en comenzar en un nodo arbitrario y decidir entonces en cada paso que vamos a visitar el más próximo de los nodos que queden por visitar. En el ejemplo, si comenzamos en el nodo 1, entonces el nodo más próximo no visitado es el nodo 2. Desde el nodo 2, el nodo más próximo no visitado es el nodo 3, y así sucesivamente. Tras visitar el último nodo, volvemos al punto de partida. El recorrido que se construye de esta manera visita los nodos 1, 2, 3, 5, 4, 6 y 1, y tiene una longitud total de 60. Por tanto, aunque el algoritmo voraz no encuentra una solución óptima en este caso, tampoco se equivoca demasiado. Con otros ejemplos, sin embargo, puede ser catastrófico; véase el problema 13.4. ¿Es posible encontrar un algoritmo aproximado del cual se garantice que encontrará una solución razonablemente buena? Vamos a ver que la respuesta es si, si restringimos la clase de casos que se consideran (Sección 13.2.1), y que probablemente es no en caso contrario (Sección 13.3.2).

13.2 ALGORITMOS APROXIMADOS

13.2.1 El viajante métrico

Tal como veremos en la Sección 13.3.2, es imposible encontrar un buen algoritmo aproximado para el problema del viajante a no ser que sea P = NP. Aquí describiremos un algoritmo aproximado eficiente para un caso especial. Se dice que una

matriz de distancias tiene la propiedad *métrica* si es válida la desigualdad triangular: para tres ciudades cualesquiera i, j y k

$$distancia(i, j) \leq distancia(i, k) + distancia(k, j)$$

En otras palabras, la distancia directa desde i hasta j nunca es mayor que la distancia desde i hasta j pasando por k. En particular, esta propiedad es válida en los problemas *euclídeos*, en los cuales se considera que las ciudades yacen en un plano y que las distancias son las distancias en línea recta entre ellas. Cuando al hablar de «distancia» nos referimos realmente a un coste, la propiedad métrica puede surgir de forma menos natural. En el ejemplo anterior, la matriz de distancias posee ciertamente la propiedad métrica. Por ejemplo:

$$10 = distancia(1,3) \leq distancia(1, 2) + distancia(2, 3) = 3 + 8 + 11,$$

y se cumple una condición similar para toda selección de tres nodos.

Sea G un grafo completo no dirigido con n nodos, y consideremos cualquier ciclo hamiltoniano de este grafo. Supongamos que el ciclo tiene longitud h. Claramente, consta de n aristas. Si eliminamos alguno de ellos, los $n-1$ aristas restantes forman una ruta que visita todos los nodos de G una sola vez, pero no vuelve al punto de partida: esto es lo que se denomina un camino *hamiltoniano*. Dado que el arista eliminado tiene una longitud no negativa, la longitud de esta ruta hamiltoniano es h como máximo. Sin embargo, el camino Hamiltoniano es al mismo tiempo un árbol que abarca al grafo G. Si la longitud de un árbol de expansión[1] mínimo para G es $m(G)$, se sigue que el camino hamiltoniano tiene que tener una longitud mayor o igual que $m(G)$. Por tanto, para todo ciclo hamiltoniano de G, $h \geq m(G)$.

Supongamos ahora que la matriz de distancias G tiene la propiedad métrica. Vamos a ilustrar la forma en que funciona el algoritmo aproximado empleando la matriz de distancias dada anteriormente. En primer lugar, se halla un árbol de expansión mínimo empleando o bien el algoritmo de Kruskal o bien el de Prim; véanse las Secciones 6.3.1 y 6.3.2. La figura 13.3 muestra un árbol de expansión mínimo para nuestro ejemplo. Se ha dibujado con el nodo 1 en la raíz, puso que nos interesan aquellos recorridos que empiezan y acaban en este nodo (pero véase el problema 13.8). El árbol de expansión mínimo tiene una longitud 34. Imagine ahora que usted fuera una hormiga que fuera recorriendo el contorno de esta figura. Comenzando en la raíz (nodo 1), baje por el lado izquierdo de la arista {1, 2} hasta el nodo 2, dé la vuelta al nodo 2, vuelva a subir por el lado derecho de la arista {1, 2}, baje por el lado izquierdo de la arista {1, 5} hasta el nodo 5, pase por el nodo 5 y baje por el lado izquierdo de la arista {5, 3} hasta el nodo 3, dé la vuelta al nodo 3 y así sucesivamente. Eventualmente, volverá a la raíz después de subir por el lado derecho de las aristas {4, 6}, {4, 5} y {5, 1}. La línea discontinua de la figura muestra el recorrido completo.

Dado que el árbol abarca al grafo subyacente, estamos seguros de que visitaremos todos los nodos al menos una vez durante el recorrido. De hecho, tal como

muestra el ejemplo, quizá se visiten algunos nodos más de una vez: el recorrido completo de la figura visita los nodos 1, 2, 1, 5, 3, 5, 4, 6, 4, 5 y 1 por este orden. Llamemos t_0 a este recorrido. Está claro que durante el recorrido se pasa dos veces por todas las aristas del árbol, una vez por el lado izquierdo y otra por el lado derecho. Si la longitud del árbol de expansión (o de envergadura) mínimo es $m(G)$, por tanto, la longitud $lon(t_0)$ es $2m(G)$.

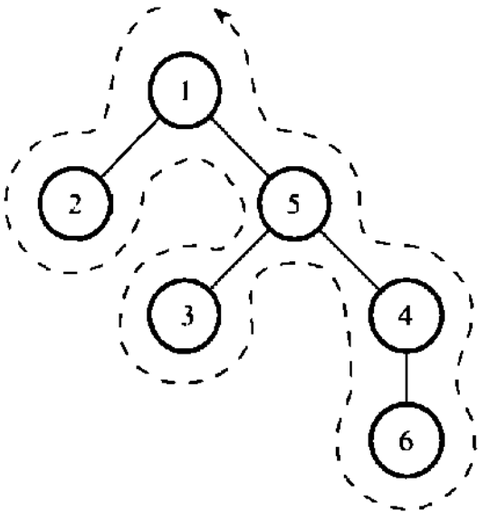


Figura 13.3. Un árbol de expansión mínimo

El algoritmo aproximado procede ahora a cortar los nodos duplicados en el recorrido. En el ejemplo, el primer nodo que se vuelve a visitar es el 1, que se visita por segunda vez entre los nodos 2 y 5. Acortamos nuestro recorrido omitiendo esta segunda visita al nodo 1.

El nuevo recorrido es 1, 2, 5, 3, 5, 4, 6, 4, 5, 1. Si llamamos t_1 a este recorrido, entonces

$$lon(t_0) - lon(t_1) = distancia(2,1) + distancia(1,5) - distancia(2,5)$$

que debe de ser mayor o igual que cero por la propiedad métrica de la matriz de distancias. Por tanto, t_1 no es mayor que t_0. En t_1 el nodo 5 se visita de nuevo entre los nodos 3 y 4. Tal como antes, podemos omitir la segunda visita al nodo 5, obteniendo un nuevo nodo t_2 que visita los nodos 1, 2, 5, 3, 4, 6, 4, 5, 1. Un argumento similar al del caso anterior demuestra que es $lon(t_2) \leq lon(t_1)$. Procediendo de esta manera, se van omitiendo sucesivamente del recorrido aquellos nodos que hayan sido visitados previamente (salvo por el retorno final al nodo 1); cada una de las rutas obtenidas es como máximo tan larga como su predecesora. En el ejemplo, el camino final obtenida, a la que llamaremos sencillamente t, visita los nodos 1, 2, 5, 3, 4, 6 y 1. Su longitud es 64. En general, dado que $lon(t) \leq lon(t_0) = 2m(G)$ y que la longitud h de todo ciclo hamiltoniano de G es al menos $m(G)$, tenemos que

$$lon(t) \leq 2m(G) \leq 2h$$

La longitud del camino hallada mediante este algoritmo aproximado no es, por tanto, mayor que el doble de la longitud del camino óptimo. En el ejemplo, la longitud del camino óptimo es al menos 34, y como máximo 65.

Aun cuando la demostración de que el algoritmo funciona nos ha obligado a obtener t de forma poco directa, es fácil ver que de hecho t es simplemente una lista de los nodos de un árbol de recubrimiento mínimo de G en preorden; véase la Sección 9.2. La implementación del algoritmo es sencilla. Para un grafo con n nodos, la búsqueda de un árbol de recubrimiento mínimo requiere un tiempo que está en $\Theta(n^2)$, si utilizamos el algoritmo de Prim; véase la Sección 6.3.2. La exploración de este árbol en preorden requiere un tiempo que está en $\Theta(n)$. El algoritmo aproximado, por tanto, requiere un tiempo que está en $\Theta(n^2)$.

Empleando un algoritmo aproximado más sofisticado, podemos garantizar que llegaremos a un factor $\frac{1}{2}$ de la solución óptima; véase el problema 13.10. Hay muchas variaciones sobre el tema del viajante. Por ejemplo, el grafo puede ser dirigido, así que *distancia*(i, j) no es necesariamente igual a *distancia*(j, i), o quizá falten algunos de las aristas. No estudiaremos aquí estas variaciones, salvo para indicar que para el caso de un grafo dirigido, no se conoce ninguna heurística con un rendimiento garantizado en el caso peor, ni siquiera en el caso métrico.

13.2.2 El problema de la mochila (5)

Ya nos hemos encontrado con distintas variantes del problema de la mochila a lo largo del libro. Recuerde que nos dan n objetos y una mochila. Para cada i, $1 \leq i \leq n$, el objeto i tiene un peso entero positivo w_i y un valor entero positivo v_i. La mochila puede transportar un peso total que no sobrepase W. Nuestro objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos incluidos, respetando la limitación de capacidad. En la Sección 6.5 permitíamos que los objetos se fragmentaran: podíamos llevar un tercio del objeto i, por ejemplo, con unos beneficios de $v_i/3$ y un coste $w_i/3$. En este caso, veíamos que un sencillo algoritmo voraz hallaba la solución óptima en un tiempo que está en $O(n \log n)$: basta con seleccionar los objetos por orden no creciente de valores por unidad de peso y llenar la mochila totalmente, fragmentando el último objeto considerado si fuera necesario.

En la Sección 8.4 abordamos el problema más difícil consistente en hallar una solución óptima cuando no se permite fragmentar objetos: podemos llevarnos un objeto o dejarlo atrás, pero no podemos llevarnos un fragmento de un objeto. En este caso vimos que el algoritmo voraz puede no ser óptimo, lo cual no es sorprendente porque esta versión del problema es de dificultad NP. Sin embargo, vimos un algoritmo de programación dinámica que halla una solución óptima en un tiempo que está en $\Theta(nW)$, lo cual puede ser prohibitivo cuando W es grande. Por último, vimos una tercera variante del tema en las Secciones 9.6.1 y 9.7.2. Esto no nos concierne ahora, pero véase el problema 13.12.

Aunque es subóptimo si no se permite la fragmentación de objetos, el algoritmo voraz es tan eficiente en términos de tiempo de cálculo que puede ser útil si se

garantiza que su error relativo estará bajo control. Para ser precisos, enunciaremos explícitamente este algoritmo:

```
función mochila-voraz(w[1..n], v[1..n],W)
    Se ordena el caso de modo que v[i]/w[i]≥v[j]/w[j] para todo 1 ≤ i ≤ j ≤ n
    peso, valor ← 0
    para i ← 1 hasta n hacer
        si peso + w[i] ≤ W entonces   valor ← valor + v[i]
                                      peso ← peso + w[i]
    devolver valor
```

Desafortunadamente, este algoritmo puede ser arbitrariamente malo. Para ver esto, tomemos un entero $x > 2$ tan grande como se desee, y consideremos el caso que consiste en una mochila de capacidad $W = x$ y dos objetos cuyos pesos y valores son $w_1=1$, $v_1=2$, $w_2=x$ y $v_2=x$. Los objetos están en orden no creciente de valores por unidad de peso, porque $v_1 / w_1=2$ es mayor que $v_2 / w_2=1$. El algoritmo pone primero el objeto 1 en la mochila, porque $w_1 \leq W$; esto produce el valor 2. Como resultado, el segundo objeto ya no cabe en la mochila, y el algoritmo proporciona el 2 como valor aproximado. Por supuesto, la solución óptima en este caso es abandonar el objeto 1 y meter en su lugar el objeto 2, con un rendimiento x. Por tanto, el algoritmo voraz proporciona un valor que está dentro de un factor $x/2$ del óptimo, que se puede hacer tan malo como se desee.

Afortunadamente, esto es fácil de arreglar. Considere esta pequeña modificación del algoritmo voraz, que supone por sencillez que no hay objetos cuyo peso supere la capacidad de la mochila:

```
función mochila-aprox(w[1..n], v[1..n], W)
    máximo ← máx{v[i] | 1 ≤ i ≤ n}
    devolver máx(máximo, mochila-voraz(w, v, W))
```

Ahora demostraremos que la solución que proporciona *mochila-aprox* está siempre dentro de un factor 2 de la solución óptima. Considere un caso arbitrario. Suponga que la mochila es demasiado pequeña para contener todos los objetos a la vez, dado que en caso contrario el algoritmo voraz proporciona la solución óptima trivial que incluye a todos los objetos. Suponga también que los objetos ya están ordenados por orden no creciente de peso unitario. Sean *opt* y $\widetilde{opt}$ la solución óptima y la solución proporcionada por *mochila-aprox*, respectivamente. Sea ℓ el menor entero tal que $\sum_{i=1}^{\ell} w_i > W$. Este ℓ existe por hipótesis. Considere un caso modificado del problema de la mochila que utiliza los mismos objetos pero para el cual la capacidad de la mochila se aumenta hasta $W' = \sum_{i=1}^{\ell} w_i$. La demostración del Teorema 6.5.1 es aplicable *mutatis mutandis* para demostrar que para el caso modificado resulta óptimo incluir los ℓ primeros objetos. La solución de este caso es, por tanto, $opt' = \sum_{i=1}^{\ell} v_i$. Pero la solución del caso original no puede ser mayor que

la del caso modificado, porque no se puede acumular un valor mayor en una mochila más pequeña cuando disponemos de los mismos objetos: $opt \leq opt'$. Queda por tener en cuenta que *mochila-voraz*$(w, v, W) \geq \sum_{i=1}^{\ell-1} v_i$ porque el algoritmo voraz pondrá los $\ell-1$ primeros objetos en la mochila antes de dejar sin meter el l-ésimo. (Quizá añada algunos objetos más). Además, *máximo* $\geq v_\ell$, en donde *máximo* es el mayor de los v_i según se ha calculado en *mochila-aprox*. Resumiendo, y empleando el hecho de que $\max(x,y) \geq (x+y)/2$, obtenemos finalmente

$$\begin{aligned}
\widetilde{opt} &= \max(máximo, mochila - voraz(w, v, W)) \\
&\geq (máximo + mochila - voraz(w, v, W)) / 2 \\
&\geq \left(v_\ell + \sum_{i=1}^{\ell-1} v_i \right) / 2 = \sum_{i=1}^{\ell} v_i / 2 = opt' / 2 \\
&\geq opt / 2
\end{aligned}$$

lo cual demuestra que la solución aproximada está dentro de un factor 2 de la óptima, tal como deseábamos. Veremos en la Sección 13.5 que se pueden obtener eficientemente aproximaciones aún mejores para el problema de la mochila.

13.2.3 Llenado de cajas

De forma bastante parecida al problema de la mochila, nos dan n objetos. El objeto i tiene un peso w_i, con $1 \leq i \leq n$. También se nos da un cierto número de cajas idénticas, cada una de las cuales puede contener cualquier número de objetos siempre y cuando su peso total no supere la capacidad W de la caja. Los objetos no se pueden fragmentar. Surgen dos problemas relacionados. Dadas k cajas, ¿cuál es el mayor número de objetos que se pueden almacenar? Y por otra parte, ¿cuál es el menor número de cajas que se necesitan para almacenar todos los objetos? El segundo problema suele llamarse *problema de llenado de cajas*. Ambos problemas son de dificultad NP.

Comenzaremos por examinar el primer problema. Supongamos, sin pérdida de generalidad, que los objetos que hay que almacenar están numerados por orden de pesos no decrecientes, así que $w_i \leq w_j$ cuando $i \leq j$. Para el caso sencillo en que k=1, resulta evidente que es óptimo cargar los objetos en la única caja disponible por orden numérico. Consideremos ahora el siguiente algoritmo voraz sencillo para el caso en que k=2. Vamos a ir tomando cada objeto, por orden numérico. Ponemos tantos como sea posible en la caja 1; cuando esté llena la caja 1, ponemos tantos como sea posible de los objetos restantes en la caja 2; finalmente, nos detenemos.

Basta un sencillo ejemplo para mostrar que este algoritmo no siempre da la solución óptima para nuestro problema. Por ejemplo, si n=4, si los pesos de los cuatro objetos son 2, 3, 4 y 5 respectivamente, y la capacidad de las cajas es 7, entonces el algoritmo voraz pone los objetos 1 y 2 en la caja 1, y el objeto 3 en la caja 2, y se detiene. De esta manera, ha conseguido almacenar sólo 3 objetos en las dos

cajas. La solución óptima es, evidentemente, poner los objetos 1 y 4 en la caja 1, y los objetos 2 y 3 en la caja 2, totalizando así 4 objetos. Sin embargo, el algoritmo voraz nunca se equivoca en más de 1 objeto, tal como mostraremos a continuación.

Sea s la solución óptima para el caso de dos cajas, esto es, supongamos que se pueden almacenar s objetos en las dos cajas. Pero supongamos primero que en lugar de tener dos cajas de capacidad W cada una, tenemos solamente una caja de capacidad $2W$. Construimos la solución óptima de este otro caso poniendo objetos en la caja por orden numérico. Supongamos que de esta manera se pueden cargar t objetos. Claramente, $s \leq t$ porque al partir una caja grande en dos nunca podremos meter más objetos que antes. En la solución óptima para el caso de una caja grande, sea j el menor índice tal que $\sum_{i=1}^{j} w_i > W$. El índice j está bien definido a no ser que $\sum_{i=1}^{n} w_i \leq W$, en cuyo caso la solución óptima trivial es poner los n objetos en la primera caja. Empleando esto, y el hecho consistente en que $\sum_{i=1}^{t} w_i \leq 2W$, obtenemos $\sum_{i=j+1}^{t} w_i < W$. La solución se ilustra en la figura 13.4.

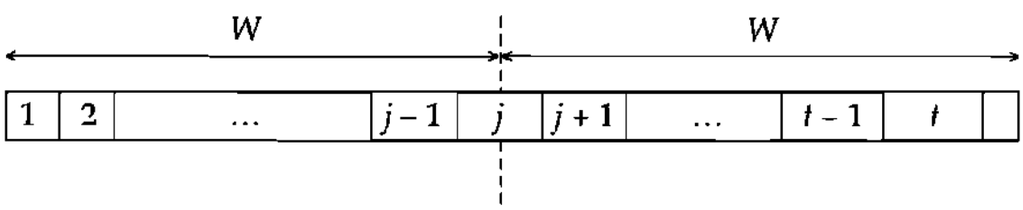


Figura 13.4. Llenado de una caja grande de capacidad 2W

Volviendo al caso de las dos cajas, es posible cargar los objetos del 1 al $j-1$ en la primera caja, y los objetos del $j+1$ al t en la segunda. Sin embargo, al estar los objetos numerados en orden decreciente, obtenemos

$$\sum_{i=1}^{j-1} w_i \leq \sum_{i=j+1}^{t} w_i,$$

por tanto, el algoritmo aproximado voraz pondrá los objetos del 1 al $j-1$ en la primera caja y los objetos j al $t-1$ en la segunda. Es posible que el objeto t también quepa en la segunda caja. Las solución que encuentra el algoritmo voraz es, por tanto, al menos $t-1$. Dado que $s \leq t$, esta solución incurre como máximo en un error de un objeto.

Si los n objetos no están ordenados inicialmente entonces se necesita un tiempo que está en $\Theta(n \log n)$ para ordenarlos. Después, el algoritmo voraz aproximado requiere un tiempo que está en $\Theta(n)$. El algoritmo voraz se puede extender en la forma evidente para al caso en que estén disponibles k cajas. En este caso, la solución que encuentre nunca incurrirá en un error de más de $k-1$ objetos. Esto se puede demostrar mediante una sencilla extensión del argumento anterior.

El segundo de los dos problemas asociados pregunta: dados n objetos, cuántas cajas se necesitan para almacenarlos todos. Resulta tentador ensayar la variante

evidente del algoritmo aproximado que se describe más arriba: tomar los objetos por orden de pesos no decrecientes, poner tantos como sea posible en la primera caja, después en la segunda, y así sucesivamente, y contar el número de cajas que se necesitan para almacenar los n objetos.

Sea b el número óptimo de cajas que se necesitan, y sea s la solución hallada por este algoritmo aproximado. Esta vez, no es cierto que el error absoluto $s-b$ esté acotado por una constante; sin embargo, es cierto que s es menor que un múltiplo constante de b. En otras palabras, s/b está acotado. Para ilustrar la imposibilidad de obtener una cota de error que sea constante aditiva mediante este algoritmo aproximado, considere la siguiente familia de casos.

Nos dan $n = 2k$ objetos y un suministro de cajas de capacidad $2W$. Resulta cómodo suponer que k es par. El valor exacto de W es irrelevante siempre y cuando sea mayor que $3(k-1)$. Los n objetos tienen los pesos $W-k$, $W-(k-1)$, ...,$W-1$, $W+1$,...,$W+(k-1)$, $W+k$. Claramente, los n objetos se pueden almacenar en k cajas tomando el más ligero junto con el más pesado, el siguiente más ligero con el siguiente más pesado, y así sucesivamente. El algoritmo voraz, por otra parte, pone primero el objeto de peso $W-k$ en una caja junto con el objeto cuyo peso es $W-(k-1)$. El tercer objeto ya no puede ir a la misma caja, porque

$$W-k + W-(k-1) + W-(k-2) = 3W-3\,(k-1) > 2W.$$

Por tanto, los objetos cuyo peso es menor que W se almacenan de dos en dos en $k/2$ cajas, mientras que los k objetos restantes ocupan una caja cada uno, siendo el total de $3k/2$ cajas. En este caso, la diferencia entre la solución óptima $b = k$ y la solución $s = 3k/2$ hallada por el algoritmo aproximado se puede hacer tan grande como queramos sin más que seleccionar un k suficientemente grande. Por otra parte, para esta familia de casos la razón s/b es constante.

En general, se puede demostrar que para este algoritmo voraz aproximado

$$s \leq 2+\frac{17}{10}\,b$$

Se obtiene un algoritmo aproximado mejor si los objetos se consideran por orden de peso no *creciente*. Ahora se va tomando cada objeto por turno, y se intenta añadir el objeto a la caja 1; si no cabe, intentamos añadirlo a la caja 2, y así sucesivamente; si no cabe en ninguna de las cajas utilizadas hasta el momento, empezamos una caja nueva que tenga el número inmediato superior. Observe que para el caso discutido en el párrafo anterior, este algoritmo halla el llenado óptimo. En general, para este algoritmo voraz aproximado, $s \leq 4+\frac{11}{9}\,b$. Las demostraciones de las cotas que se dan en este párrafo no son sencillas.

13.3 PROBLEMAS DE APROXIMACIÓN CON DIFICULTAD NP

Quizá el Capítulo 12 le haya producido la impresión de que no hay una diferencia significativa entre la dificultad de resolver uno u otro problema NP- comple-

to. Nada más lejos de la realidad cuando se trata de hallar soluciones aproximadas. Por ejemplo, vimos en la Sección 13.2.3 que aun cuando la determinación del mayor número de objetos que se pueden almacenar en dos cajas es de dificultad NP, un sencillo algoritmo voraz halla rápidamente una solución que nunca dista de ser óptima por valor de más de un elemento. Y sin embargo, otros problemas de optimización siguen siendo de dificultad NP aun cuando nos contentemos con mantener bajo control el error *relativo*. Por ejemplo, dado cualquier $\alpha > 1$, resulta ser de dificultad NP la búsqueda de un recorrido para el problema del viajante tal que esté garantizado que su longitud se encuentre dentro de un factor α de la óptima; véase la Sección 13.3.2. Hay otros problemas intermedios: el problema del viajante *métrico* se puede resolver eficientemente salvo un factor 2 del valor óptimo mediante el algoritmo que se vio en la Sección 13.1.2 —y existen mejores algoritmos aproximados (véase el problema 13.10)— pero ningún algoritmo aproximado eficiente puede garantizar una cota superior fija para el valor absoluto de sus soluciones a no ser que P = NP; véase la Sección 13.3.1. Un ejemplo espectacular de la forma en que el mismo problema de optimización puede dar lugar a dos problemas de aproximación muy distintos se presenta en la Sección 13.4.

Considere un problema de optimización y sea $opt(X)$ el valor de una solución óptima del caso X. Por ejemplo, si consideramos el problema de coloreado de grafos de la Sección 13.1.1 y si G es un grafo, entonces $opt(G)$ denota el número cromático de G: el menor número de colores que bastan para colorear los vértices de G de tal manera que no se asigne el mismo color a ninguna pareja de vértices adyacentes. Para el caso X, un algoritmo aproximado hallaría algún valor $\widetilde{opt}(X)$ que podría ser subóptimo, pero que se requiere que sea *factible*. Por ejemplo, si el grafo G se puede colorear con cinco colores pero no menos, entonces $opt(G)=5$ y un algoritmo que proporcione $\widetilde{opt}(G)= 7$ será subóptimo pero aceptable, porque *es* posible colorear G con siete colores, siempre y cuando existan al menos otros tantos vértices. Un algoritmo que proporcionase $\widetilde{opt}(G)=4$, por otra parte, sería incorrecto, porque G no se puede colorear con cuatro colores. En la mayoría de los casos, el requisito de factibilidad corresponde al requisito $\widetilde{opt}(X) \geq opt(X)$ para problemas de minimización, y a $\widetilde{opt}(X) \leq opt(X)$ para problemas de maximización. En la práctica, quizá deseemos la solución óptima o aproximada, en lugar de su coste: quizá necesitemos una asignación real de colores a los nodos de G que no emplee más de $\widetilde{opt}(G)$ colores. Sin embargo, estos problemas suelen ser equivalentes en virtud de la reducibilidad a decisión: véase el problema 12.43.

Sean c y ε dos constantes positivas. A todo problema de optimización P le corresponden problemas de aproximación absoluta y relativa. Supongamos por sencillez que todas las soluciones factibles para casos del problema P son estrictamente positivas. El problema de *aproximación absoluta* c, que se denota c-abs-P, es el problema consistente en hallar, para cualquier caso X, una solución factible $\widetilde{opt}(X)$ cuyo error absoluto en comparación con la solución óptima $opt(X)$ sea como máximo c:

$$opt(X) \leq \widetilde{opt}(X) \leq opt(X) + c \text{ o bien}$$

$$opt(X) - c \leq \widetilde{opt}(X) \leq opt(X),$$

para problemas de minimización y maximización, respectivamente. El problema de *aproximación relativa* ε, que se denota ε–rel-P, es el problema de hallar, para cualquier caso X, una solución factible cuyo error relativo, comparado con la solución óptima, sea como máximo ε:

$$opt(X) \leq \widetilde{opt}(X) \leq (1 + \varepsilon)opt(X) \text{ o bien}$$

$$(1 - \varepsilon)opt(X) \leq \widetilde{opt}(X) \leq opt(X),$$

para problemas de minimización y maximización, respectivamente. Observe que $\varepsilon \geq 1$ sólo es interesante para problemas de minimización, a no ser que sea difícil hallar una solución factible incluso sin la limitación de que sea óptima.

Por ejemplo, hemos visto algoritmos eficientes para el problema de aproximación 1-absoluta correspondiente a determinar el mayor número de objetos que se pueden almacenar en dos cajas, y para el problema aproximado 1-relativo del viajante métrico. Ahora veremos que este último problema es más difícil de aproximar que el primero, en el sentido de que el problema del viajante métrico aproximado c-absoluto es exactamente igual de difícil que el problema exacto, independientemente de lo grande que estemos dispuesto seleccionar c. El problema del viajante sin limitaciones es todavía más difícil de aproximar: incluso el problema del viajante aproximado ε relativo es tan difícil como el problema exacto, independientemente de lo grande que seleccionemos ε. Para demostrar estos resultados, empleamos la noción de reducciones polinómicas que se vio en la Sección 12.5.2 para demostrar la forma en que se podría resolver eficientemente el problema exacto si fuéramos capaces de resolver eficientemente el correspondiente problema de aproximación.

13.3.1 Problemas de aproximación con dificultad absoluta

Denotaremos el problema del viajante métrico con las siglas MTSP. Supongamos por sencillez que las distancias entre ciudades son números enteros. En la Sección 13.1.2 vimos un algoritmo eficiente para 1-rel-MTSP. Ahora demostraremos que MTSP $\leq_T^P$ c – *abs* – MTSP para cualquier constante positiva c. Para esto, suponga que dispone de algún algoritmo para resolver el problema del vendedor métrico c absoluto. Tenemos que encontrar un algoritmo para el problema exacto del vendedor métrico que sea eficiente, siempre y cuando no tengamos en cuenta el tiempo invertido empleando el algoritmo aproximado. Por tanto, todo algoritmo eficiente para el problema aproximado se traducirá a un algoritmo eficiente para el problema exacto. Suponiendo que el problema exacto del viajante métrico sea realmente difícil (lo cual es más que probable, puesto que es de dificultad NP), concluimos que no puede existir ningún algoritmo eficiente para poder hallar aproximaciones c absolutas del problema del viajante métrico.

Teorema 13.3.1. MTSP $\leq_T^p$ $c-abs-$MTSP *para cualquier constante positiva* c.

Demostración Sea c una constante positiva y consideremos un algoritmo arbitrario para resolver c-abs-MTSP, el problema del viajante métrico aproximado c-absoluto. Considere un caso de MTSP representado por una matriz simétrica $n \times n$ que respete la desigualdad triangular. Sea $opt(M)$ la longitud de un recorrido óptimo en este caso. Construimos un nuevo caso M' multiplicando todas las entradas de M por $\lfloor c \rfloor+1$. Está claro que M' también satisface la desigualdad triangular y que es simétrica; por tanto, define un caso legítimo de MTSP. Está igualmente claro que todo recorrido óptimo de las ciudades según la matriz M es también óptimo según M', y viceversa, salvo que la longitud del recorrido será $\lfloor c \rfloor+1$ veces mayor según M'. Por tanto, $opt(M') = (\lfloor c \rfloor+1)opt(M)$. Considere ahora el resultado $\widetilde{opt}(M')$ de ejecutar nuestro supuesto algoritmo aproximado c absoluto en la matriz M'. Por definición:

$$\begin{aligned}(\lfloor c \rfloor+1)opt(M) = opt(M') \leq \widetilde{opt}(M') &\leq opt(M')+c \\ &= (\lfloor c \rfloor+1)opt(M)+c < (\lfloor c \rfloor+1)(opt(M)+1)\end{aligned}$$

Dividiendo por $\lfloor c \rfloor+1$, se obtiene

$$opt(M) \leq \frac{\widetilde{opt}(M')}{\lfloor c \rfloor+1} < opt(M)+1$$

Concluimos que $opt(M) = \lfloor \widetilde{opt}(M')/(\lfloor c \rfloor+1) \rfloor$, porque $opt(M)$ es entero. Entonces la solución de MTSP se obtiene fácilmente a partir de cualquier solución aproximada c absoluta del caso M' de MTSP, lo cual completa la reducción.

Hay muchos otros problemas para los cuales hallar soluciones aproximadas c absolutas tiene dificultad NP, independientemente de lo grande que sea c. Entre éstos se cuentan el problema de la mochila y el problema de la camarilla máxima; véanse los problemas 13.15 y 13.16. Esto sucede con todos aquellos problemas que admiten lo que se denomina «escalado creciente»: si es posible transformar eficientemente cualquier caso en otro cuya solución óptima sea $\lfloor c \rfloor+1$ veces mayor, y si la solución óptima es un entero positivo, entonces es igualmente difícil hallar soluciones aproximadas c-absolutas y hallar soluciones óptimas.

13.3.2 Problemas de aproximación con dificultad relativa

Denotaremos el problema del viajante sin restricciones (por oposición al métrico) mediante TSP. (En la Sección 12.5.2 se utilizaba TSP para denotar el problema de hallar un recorrido óptimo en lugar de determinar su longitud; esto es admisible porque se puede ver fácilmente que ambas versiones del problema son polinómicamente equivalentes.) No sólo es tan difícil hallar aproximaciones absolutas de este problema como hallar soluciones exactas, sino que además esto es cierto también para las aproximaciones relativas. Simbólicamente, TSP $\leq_T^p$ $\varepsilon-rel-$ TSP para cual-

quier constante positiva ε. En lugar de demostrar esto directamente, mostraremos que las aproximaciones ε-relativas de TSP pueden servir para resolver el problema de decisión del ciclo hamiltoniano HAMD que se encontraba en la Sección 12.5.2. Esta técnica es similar a la que se utilizaba en el teorema 12.5.9 para demostrar que la solución *exacta* de TSP se puede utilizar para resolver HAMD. La conclusión deseada se incluye como corolario 13.3.3 porque el problema del ciclo hamiltoniano es NP completo.

Teorema 13.3.2. $\text{HAMD} \leq_T^p \varepsilon - rel - \text{TSP}$ *para cualquier constante positiva* c.

Demostración. Sea ε una constante positiva y consideremos un algoritmo cualquiera para resolver ε–rel-TSP, el problema aproximado ε–relativo del viajante. Considere un caso del problema hamiltoniano de decisión HAMD dado por un grafo $G = <N, A>$. Sea n el número de nodos de G, y supongamos sin pérdida de generalidad que $N = \{1, 2, \ldots, n\}$. Se construye un caso del problema del viajante en la forma siguiente:

$$M_{ij} = \begin{cases} 1 & \text{si } \{i,j\} \in A \\ 2 + \lfloor n\varepsilon \rfloor & \text{en caso contrario} \end{cases}$$

Sea $opt(M)$ una solución óptima del problema del viajante M, y sea $\widetilde{opt}(M)$ la aproximación proporcionada por nuestro supuesto algoritmo aproximado ε relativo. Por definición:

$$opt(M) \leq \widetilde{opt}(M) \leq (1+\varepsilon)opt(M).$$

Hay dos casos.

◊ Si existe un ciclo hamiltoniano en G, este ciclo define un recorrido para el viajante que solo utiliza aristas de longitud 1. Por tanto hay una solución de longitud n, que es claramente óptima. En este caso,

$$\widetilde{opt}(M) \leq (1+\varepsilon)opt(M) = (1+\varepsilon)n$$

◊ Si en G no hay ciclos hamiltonianos, todo recorrido del viajante debe de utilizar al menos una arista de longitud $2+\lfloor n\varepsilon \rfloor$, además de $n-1$ aristas de longitud 1 como mínimo cada uno, con una longitud total de al menos $2+\lfloor n\varepsilon \rfloor + (n-1) > (1+\varepsilon)n$. Por tanto:

$$\widetilde{opt}(M) \geq opt(M) > (1+\varepsilon)n.$$

De esa manera, podemos decidir si en G hay o no un ciclo hamiltoniano examinando la respuesta proporcionada por el algoritmo para el problema del viajante: hay un ciclo hamiltoniano si y solo si $\widetilde{opt}(M) \leq (1+\varepsilon)n$. Esto completa la reducción.

Obsérvese que el caso del problema del viajante construido en la demostración anterior *no* es métrico. Si en G hay una arista entre los vértices i y k, y entre los vértices k y j pero no entre los vértices i y j, y si $n \geq 1/\varepsilon$:

$$distancia(i, j) = 2+\lfloor n\varepsilon \rfloor > 2 = distancia(i, k) + distancia(k, j)$$

Eso era inevitable, dado que 1-rel-MTSP se puede resolver eficientemente.

Corolario 13.3.3 $TSP \leq_T^p \varepsilon - rel - TSP$ *para cualquier constante positiva* ε.

Demostración. Sin duda recordará que, inmediatamente antes del teorema 12.5.9, TSPD denota el problema de *decisión* del viajante. Sabemos por el problema 12.47 que el problema del viajante es reducible al problema de la decisión en el sentido de que $TSP \equiv_T^p TSPD$. Por otra parte, sabemos por el problema 12.50 que el problema del ciclo hamiltoniano es NP completo. Por definición de NP-completitud y partiendo del hecho evidente consistente en que TSP pertenece a NP, se sigue que $TSP \leq_T^p HAMD$. Acabamos de mostrar que $HAMD \leq_T^p \varepsilon - rel - TSP$. Alcanzamos la conclusión deseada por la transitividad de las reducciones polinómicas.

Hay muchos otros problemas para los cuales la búsqueda de soluciones aproximadas NP relativas ε tiene dificultad NP, independientemente de lo grande que admitamos que puede ser ε (respetando $\varepsilon < 1$ en los problemas de maximización). Entre éstos se cuentan el problema del racimo mínimo (véase la Sección 13.4, más adelante), el problema de camarilla máxima y el problema de búsqueda del número cromático de un grafo. De hecho, se sabe que estos dos últimos problemas son todavía más difíciles de aproximar que éste: suponiendo que $P \neq NP$, ningún algoritmo de tiempo polinómico puede hallar una camarilla dentro de un grafo de n nodos con la garantía de que esté dentro de un factor $\sqrt[6]{n}$ del valor óptimo para $\delta > 6$, y lo mismo sucede para el coloreado óptimo de grafos con $\delta > 14$. Por otra parte, existen problemas de optimización tales como el problema del vendedor métrico para los cuales es sencillo hallar aproximaciones ε relativas para un ε suficientemente grande, y sin embargo el problema de aproximación llega a tener dificultad NP cuando ε es demasiado pequeño; véase la Sección 13.5 para ejemplos adicionales.

13.4 LO MISMO, PERO DISTINTO

Sea $G = \langle N, A \rangle$ un grafo no dirigido, y sea $c : A \rightarrow \mathbb{R}^+$ una función de coste. Considere una partición de N en tres subconjuntos: N_1, N_2 y N_3, que en lo sucesivo se

llamarán *racimos*, de tal manera que cada nodo de N pertenezca exactamente a uno de los racimos. Para cada nodo u, sea *conjunto*(u) el único i tal que $u \in N_i$. Esto desglosa las aristas de G en aquéllos que enlazan nodos de un mismo racimo, que llamaremos aristas *internas*, y aquellos que enlazan nodos de racimos distintos, llamados aristas *cruzadas*. Nuestro problema de optimización consiste en seleccionar N_1, N_2 y N_3 de tal manera que el coste total de las aristas cruzadas sea máximo, o equivalentemente, de tal forma que se minimice el coste total de las aristas internas. Este problema tiene dificultad NP (es una consecuencia inmediata del teorema 13.4.2, que se da más adelante.) La figura 13.5 muestra un ejemplo en que la solución óptima consiste en formar los racimos N_1= {*a*, *b*}, N_2={*c*, *d*} y N_3= {*e*}, de tal manera que el coste total de las aristas internas es 4 y el de las aristas cruzadas es 31.

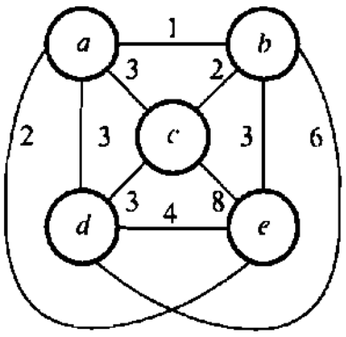


Figura 13.5. Los problemas MIN-RACIMO/MAX-CORTAR

Aunque sea equivalente maximizar el coste total de las aristas cruzadas o minimizar el coste total de las aristas internas, considere los dos problemas de optimización siguientes.

◊ MAX-CORTAR es el problema de maximizar el coste total de las aristas cruzadas para todas las particiones de N.
◊ MIN-RACIMO es el problema de minimizar el coste total de los aristas internas para todas las particiones de N.

Sorprendentemente, estos dos problemas idénticos de optimización pasan a ser completamente diferentes cuando se consideran como problemas de aproximación: el problema ε-rel-MAX-CORTAR se puede resolver eficientemente para $\varepsilon \geq 1/3$, mientras que el problema ε–rel-MIN-RACIMO sigue siendo de dificultad NP para todos los valores positivos de ε.

Teorema 13.4.1. *Es posible calcular eficientemente las aproximaciones $1/3$-relativas del problema* MAX-CORTAR.

Demostración. Sea $G = \langle N, A \rangle$ un grafo, y sean $c : A \to \mathbb{R}^+$ una función de coste. Considere el siguiente algoritmo voraz aproximado. Inicialmente, N_1, N_2 y N_3 están vacíos; formarán una partición de N cuando concluya el algoritmo. Considere cada nodo de G por turno. Añádalo al racimo que dé lugar al menor incremento de coste de las aristas internas, y por tanto el mayor incremento de coste de las aristas cruzadas. En el grafo de la figura 13.5, el algoritmo pone primero los nodos a, b y c en N_1, N_2 y N_3 respectivamente. Después considera el nodo d. Al añadirlo a N_1, N_2 o N_3 se incrementaría el coste de las aristas internas por valor de 3, 2 o 3 respectivamente: por tanto, el algoritmo lo añade al racimo N_2, que pasa a ser $\{b, d\}$. Por último, añadir el nodo d a los racimos N_1, N_2 y N_3 supondría incrementar el coste de los nodos internos por valor de 6, 7 y 8 respectivamente: por tanto, el algoritmo lo añade al racimo N_1, que pasa a ser $\{a, e\}$. La solución proporcionada por el algoritmo es $N_1 = \{a, e\}$, $N_2 = \{b, d\}$ y $N_3 = \{c\}$, cuyo coste en términos de aristas cruzados es 27, más del 87% de la solución óptima que es 31.

Antes de demostrar que el coste de las aristas cruzadas que proporciona este algoritmo nunca es menor que las dos terceras partes del óptimo, resulta útil dar un código explícito para este algoritmo aproximado. Aquí *suma* acumula el coste total de todas las aristas de G, y *racimo* acumula el coste de todas las aristas internas en la solución aproximada que seleccionemos. El coste deseado para todas las aristas cruzadas está dado por la diferencia entre *suma* y *racimo*, que se calcula al final del algoritmo.

```
función MAX-CORTAR-aprox(G = <N, A>, c : A → ℝ⁺)
    N₁, N₂, N₃ ← ø
    racimo, suma ← 0
    para todo u ∈ N hacer
    costemin ← ∞
    para i ← 1 hasta 3 hacer
        coste ← 0
        para todo v ∈ N_i hacer
            si {u, v} ∈ A entonces coste ← coste + c({u, v})
        suma ← suma+coste
        si coste < costemin entonces costemin ← coste
                                     k ← i
        N_k ← N_k ∪ {u}
        racimo ← racimo + costemin
devolver suma-racimo
```

Para cada nodo u, sean z_1, z_2 y z_3 los valores acumulados en *coste* cuando i=1, 2 y 3 respectivamente; se trata del incremento del coste total de las aristas internas en que se incurriría añadiendo u al racimo correspondiente. El algoritmo añade u al racimo N_k que minimiza z_i y añade el valor de z_k a *racimo*. Las tres z se le añaden a *suma*. Por tanto, en cada pasada por el bucle externo, el valor de *suma* se incrementa por valor de al menos tres veces el incremento de *racimo*. Dado que tanto *suma* como *racimo* reciben un valor inicial nulo, se sigue que $suma \geq 3 \times racimo$ al final del algoritmo. El coste total de las aristas cruzadas en la solución descubierta por el algoritmo es, por tanto:

$$\widetilde{opt}(G,c) = suma - racimo \geq \tfrac{2}{3} suma$$

Pero el coste total $opt(G, c)$ de las aristas cruzadas en una solución óptima no puede ser menor que el coste de las aristas cruzadas en la solución aproximada hallada por el algoritmo, y no puede ser mayor que el coste total *suma* de todos las aristas del grafo. Se sigue entonces que es

$$(1 - \tfrac{1}{3})opt'(G,c) \leq \tfrac{2}{3}suma \leq \widetilde{opt}'(G,c) \leq opt'(G,c)$$

Por definición, esto nos dice que nuestro algoritmo proporciona una aproximación $\frac{1}{3}$-relativa del problema MAX-CORTAR.

Aun cuando se garantiza que el algoritmo presentado en esta demostración funcionará bien en el problema MAX-CORTAR, puede ser arbitrariamente incorrecto para el (por demás idéntico) problema MIN-RACIMO. En el grafo de la figura 13.5, por ejemplo, encuentra el valor 27 como coste aproximado de un corte máximo, que no está mal si lo comparamos con el valor real del máximo, que es 31. Y al mismo tiempo, halla el valor 8 como coste del racimo mínimo, lo cual es el doble del mínimo verdadero. El problema 13.23 pide demostrar que este algoritmo puede ser arbitrariamente incorrecto para calcular aproximaciones del problema MIN-RACIMO, incluso en grafos de cuatro nodos. Ahora demostraremos que no es un fallo de este algoritmo: no se puede garantizar que ningún algoritmo eficiente encuentre una buena aproximación ε–relativa del problema MIN-RACIMO, a no ser que P = NP.

Teorema 13.4.2. $MIN-RACIMO \leq_T^p \varepsilon - rel - MIN - RACIMO$ *para cualquier constante positiva* ε.

Demostración. Lo que vamos a demostrar es que $3COL \leq_T^p \varepsilon$–*rel*–MIN–RACIMO, donde 3COL es el problema presentado en la Sección 15.5.4 y que consistía en decidir si un grafo dado se puede colorear con tres colores. El resultado deseado se sigue (de manera similar al corolario 13.3.3) del hecho consistente en que 3COL es NP-completo (teorema 15.5.19) y MIN-RACIMO es reducible a decisión. Dejamos los detalles para el lector.

Para demostrar que $3COL \leq_T^p \varepsilon$ *–rel–MIN–RACIMO*, sea ε una constante real positiva y consideremos un algoritmo cualquiera para resolver *ε–rel–MIN–RACIMO*. Sea $G = \langle N, A \rangle$ un grafo; desearíamos saber si se puede colorear empleando tres colores. Consideremos un grafo completo K sobre el conjunto de nodos N; en este grafo hay una arista entre u y v para toda pareja u, v de N tal que $u \neq v$. Definimos la siguiente función de coste para las aristas de K:

$$c(\{u,v\}) = \begin{cases} 1 & \text{si } \{u,v\} \notin A \\ \lceil (1+\varepsilon)n^2 \rceil & \text{si } \{u,v\} \in A \end{cases}$$

Sea $opt(K, c)$ una solución óptima para el problema del racimo mínimo en el grafo K, con la función de coste c, y sea $\widetilde{opt}(K, c)$ la aproximación proporcionada por nuestro supuesto algoritmo aproximado ε–relativo. Por definición:

$$opt(K,c) \le \widetilde{opt}(K,c) \le (1+\varepsilon)opt(K,c)$$

Hay dos casos:

◊ Si el grafo G es tricoloreable, consideremos un coloreado arbitrario en verde, azul y rojo. Formamos los racimos N_1, N_2 y N_3 como los conjuntos de nodos verdes, azules y rojos respectivamente. Por definición de tricoloreado, no hay aristas de G entre dos nodos de un mismo racimo. Por definición de la función de coste para K, los nodos internos definidos por N_1, N_2 y N_3 tienen todos ellos un coste 1. Claramente, en K hay menos de n^2 aristas, y por tanto también hay menos de n^2 aristas internas. Por consiguiente, el coste total de las aristas internas es menor que n^2. Concluimos que la solución óptima para el problema del racimo mínimo es menor que n^2, y por tanto:

$$\widetilde{opt}(K,c) \le (1+\varepsilon)opt(K,c) < (1+\varepsilon)n^2$$

◊ Si el grafo G no es tricoloreable, toda partición de N en tres racimos contiene necesariamente al menos una arista interno que pertenezca a A. Por definición de la función de coste de K, el coste de ese arista es $\lceil(1+\varepsilon)n^2\rceil$, así que la solución óptima del problema del racimo mínimo no puede ser menor que eso:

$$\widetilde{opt}(K,c) \ge opt(K,c) \ge \lceil(1+\varepsilon)n^2\rceil \ge (1+\varepsilon)n^2$$

Por tanto, podemos decidir si el grafo G es o no tricoloreable examinando la respuesta proporcionada por este algoritmo aproximado para el problema del racimo mínimo: el grafo es tricoloreable si y sólo si $\widetilde{opt}(K,c) < (1+\varepsilon)n^2$. Esto completa la reducción.

13.5 ENFOQUES DE APROXIMACIÓN

Hasta el momento, los algoritmos aproximados ε–relativos que hemos visto funcionan para un valor específico de ε. Por ejemplo, en la Sección 13.4 vimos un algoritmo aproximado $\frac{1}{3}$-relativo para el problema del corte máximo, pero no hay una manera evidente de mejorar este algoritmo para garantizar una aproximación mejor. De manera similar, vimos en la Sección 13.1.2 un algoritmo aproximado 1-relativo eficiente para el problema del viajante métrico. Aun cuando el problema 13.10 pide hallar un algoritmo mejor para el mismo problema que sea aproximado $\frac{1}{2}$-relativo, el nuevo algoritmo es completamente distinto del que vimos. Además, hallar aproximaciones ε–relativas para el problema del viajante métrico tiene dificultad NP cuando ε es suficientemente pequeño. Se ofrece otro ejemplo en el problema 13.19, según el cual resulta sencillo halla un algoritmo aproximado $\frac{1}{3}$-relativo para el problema consistente en colorear un grafo planar

con el número mínimo de colores, y sin embargo cualquier mejora de la aproximación sería tan difícil de calcular en el caso peor como la solución óptima.

Por contraste, hay problemas para los cuales se pueden obtener aproximaciones ε–relativas arbitrariamente buenas. Un *esquema de aproximación* es un algoritmo que admite como entrada, además del caso propio en sí, una cota superior ε del error relativo admisible. Aun cuando resulta natural esperar que el algoritmo trabaje más cuanto más pequeño sea ε, lo mejor es que la tolerancia se pueda reducir con un coste razonable de tiempo de cálculo. Decimos que el esquema de aproximación es *completamente polinómico* si basta un tiempo en $O(p(n,1/\varepsilon))$ en el caso peor para hallar aproximaciones ε–relativas para casos de tamaño n, donde p es algún polinomio fijo dos variables.

Hay muchos problemas de dificultad NP para los cuales no pueden existir esquemas de aproximación completamente polinómicos a no ser que P = NP, y hay otros para los cuales se sabe que existen. Daremos un ejemplo de cada situación.

13.5.1 Llenado de cajas, segunda parte

Vimos en la Sección 13.2.3 que existe un algoritmo aproximado voraz eficiente para el problema de llenado de cajas, algoritmo que garantiza una solución que no utilice más de $4+\frac{11}{9}\ opt$ cajas para almacenar los objetos, donde opt es el número óptimo de cajas que se necesitan. Sin embargo, no puede existir un esquema de aproximación completamente polinómico para este problema de dificultad NP si P ≠ NP, porque sería fácil obtener una solución óptima en tiempo polinómico a partir de este esquema. Para ver esto, suponga que existe un algoritmo *BPaprox*(w[1..n], W, ε) que funciona en un tiempo $O(p(n,1/\varepsilon))$ para algún polinomio p, y del cual se garantiza que puede hallar una aproximación ε–relativa del problema de almacenar n objetos de pesos w[1..n] en tan pocas cajas de capacidad W como sea posible. Suponga, para que el problema tenga sentido, que $w[i] \le W$ para todo i. Considere el algoritmo siguiente.

```
función BP(w[1..n], W)
   devolver ⌊PBaprox(w,W,1/(n+1))⌋
```

Claramente, el algoritmo BP funciona en un tiempo polinómico respecto al tamaño del ejemplar, puesto que en este caso $1/\varepsilon = n + 1$. Por definición de aproximaciones ε-relativas, la solución $\widetilde{opt}$ que proporciona *BPaprox*(w, W, $1/(n+1)$) tiene que ser tal que

$$opt \le \widetilde{opt} \le (1+1/(n+1))opt = opt + opt/(n+1)$$

pero $opt/(n+1) < 1$ porque ciertamente es posible almacenar un objeto en cargas separadas. Entonces $opt \le \widetilde{opt} < opt + 1$ y por tanto $opt = \lfloor \widetilde{opt} \rfloor$ puesto que opt es un entero. Esto completa la demostración de que BP halla una solución exacta en tiempo polinómico en el caso peor. Esto es imposible si P ≠ NP puesto que el problema de llenado de cajas es de dificultad NP.

13.5.2 El problema de la mochila (6)

Hagamos una última visita al problema de la mochila para mostrar que admite un esquema de aproximación completamente polinómico. Prosiguiendo desde la Sección 13.2.2, utilizamos la versión del problema presentada en la Sección 8.4. Recuerde que en la Sección 13.2.2 veíamos un algoritmo aproximado que halla una solución de la cual se garantiza que distará menos que un factor 2 del valor óptimo en un tiempo que está en $O(n \log n)$: se trata de un algoritmo aproximado $\frac{1}{2}$-relativo. Vimos también en la Sección 8.4 un algoritmo de programación dinámica que halla la solución óptima en un tiempo que está en $O(nW)$, en donde n es el número de objetos y W es el tamaño de la mochila. Ahora utilizaremos estos dos algoritmos a la vez, salvo que primero modificaremos ligeramente el algoritmo de programación dinámica.

Recuerde de la Sección 8.4 que el núcleo del algoritmo de programación dinámica es una tabla $V[1..n, 0..W]$, que tiene una fila por cada objeto disponible y una columna por cada peso, desde 0 hasta W. En la tabla, $V[i, j]$ da el valor máximo de los objetos que podemos transportar si el límite de peso es j y si solamente tomamos objetos de entre los i primeros. Esta tabla se construye entrada por entrada, empleando la regla

$$V[i,j] = \max\bigl(V[i-1,j], V[i-1, j-w_i] + v_i\bigr)$$

y la solución se halla entonces en $V[n, W]$ (regrese a la Sección 8.4 para más detalles).

Una aproximación parecida emplea una tabla $U[1..n, 0..M]$ donde M es una cota superior del valor óptimo que podemos transportar en la mochila, una vez más con una fila por cada objeto disponible, pero ahora con una columna para cada posible valor total que pueda caber en la mochila. Obtendremos la cota M ejecutando nuestro algoritmo aproximado $\frac{1}{2}$-relativo y multiplicando su respuesta por 2. (El valor de *opt'* definido en la Sección 13.2.2 cuando demostramos que nuestro algoritmo aproximado proporciona un valor que es al menos la mitad del óptimo es ligeramente mejor, y más fácil de calcular.) En esta ocasión, $U[i, j]$ nos da el *peso* mínimo de los objetos que podemos transportar para lograr exactamente el valor j si sólo tomamos objetos de entre los i primeros. Esta tabla se construye entrada por entrada, empleando la regla

$$U[i,j] = \min\bigl(U[i-1,j], U[i-1, j-v_i] + w_i\bigr)$$

Los valores que estén fuera de los límites se toman como $+\infty$ (o cualquier valor mayor que W) con la excepción de $U[0, 0]$, que se toma como 0. Intuitivamente, esta regla dice que para lograr el valor j con los i primeros objetos, podemos o no utilizar el objeto i, en cuyo caso la carga pesa al menos $U[i-1, j]$, o bien podemos añadir un objeto i a una colección de objetos entre los $i-1$ primeros cuyo valor total era $j-v_i$ y por tanto cuyo peso total era al menos $U[i-1, j-v_i]$ antes de añadir el objeto i. Una vez construida la tabla, la solución está dada por el mayor j tal que $U[n, j] \leq W$. Esta apro-

ximación requiere un tiempo que está en $O(n \log n + nM)$, puesto que se necesita un tiempo en $O(n \log n)$ para calcular la cota superior M, cada una de las nM entradas de U requiere un tiempo constante para su cálculo, y se invierte un tiempo en $O(M)$ al final, para explorar la n-ésima fila de U. La tabla U se puede utilizar también para determinar no sólo el valor de la carga óptima, sino además su composición, de forma muy parecida a la Sección 8.4.

Quizá se pregunta por qué querría nadie emplear esta aproximación, que resulta ligeramente más complicada y quizá menos natural que el algoritmo de programación dinámica de la Sección 8.4. El asunto es que este nuevo algoritmo es preferible cuando los valores son más pequeños que los pesos, porque el tiempo que requiere depende del valor total de la solución óptima, más que de la capacidad total de la mochila a efectos de peso. Tal como veremos, se puede forzar a los valores —pero no a los pesos— a que sean pequeños, siempre y cuando nos conformemos con una solución aproximada.

Considere un caso del problema de la mochila que consiste en una mochila de capacidad W y n objetos. El objeto i posee un peso entero positivo w_i y un valor entero positivo v_i. Sea ε el error relativo que estamos dispuestos a admitir, $0 < \varepsilon < 1$. Sea *opt* la (desconocida) solución óptima para este caso. Sea k una constante entera que se determinará posteriormente. A partir del caso original, creamos un nuevo caso que emplea la misma mochila y el mismo número de objetos, en que todos los objetos pesan lo mismo que en el caso anterior, y en que el valor del objeto i se ha reducido a $v'_i = v_i \div k$. (Esto puede dar lugar a objetos de valor cero, lo cual no se permite en la definición del problema de la mochila. Sin embargo esto carece de importancia, porque el algoritmo aproximado ½-relativo y el algoritmo de programación dinámica siguen funcionando en este caso). Decimos que $X \subseteq \{1,2,\ldots,n\}$ es *factible* si podemos encajar los objetos correspondientes en la mochila: $\sum_{i \in X} w_i \leq W$. Esta noción de factibilidad no depende de si estamos hablando del caso original o del modificado, puesto que no se han modificado los pesos de los objetos ni la capacidad de la mochila. Sea X^* una colección de objetos que podemos meter en la mochila y que sea óptima para el caso original: X^* es factible y $\sum_{i \in X^*} v_i = opt$. Sea X' una colección de objetos que podemos introducir en la mochila y que es óptima para el caso modificado. Finalmente, sea $\widetilde{opt} = \sum_{i \in X'} v_i$.

Dado que X' es óptima para el caso modificado y X^* es factible, se sigue que

$$\sum_{i \in X'} v_i \geq \sum_{i \in X^*} v'_i$$

De manera similar:

$$opt = \sum_{i \in X^*} v_i \geq \sum_{i \in X} v_i = \widetilde{opt} \tag{13.1}$$

puesto que X^* es óptima para el caso original, y X' es factible. Por definición de v'_i, tenemos que $v_i \geq k v'_i > v_i - k$. Reuniéndolo todo:

$$\begin{aligned}\widetilde{opt} &= \sum_{i\in X} v_i \geq k \sum_{i\in X} v_i \\ &\geq k \sum_{i\in X} v_i \geq \sum_{i\in X} (v_i - k) \\ &= \sum_{i\in X} v_i - \sum_{i\in X} k \\ &\geq opt - kn.\end{aligned}$$

Basta con seleccionar $k \leq \varepsilon\, opt\, / n$ para obtener

$$\widetilde{opt} \geq opt - \varepsilon n opt / n = (1-\varepsilon) opt \tag{13.2}$$

Las ecuaciones 13.1 y 13.2 dicen que $\widetilde{opt}$ es una aproximación ε–relativa de *opt*.

A primera vista, puede parecer que no se puede hacer una determinación correcta de k mientras no se conozca el valor de la solución óptima, pero esto no es así debido a nuestro algoritmo aproximado ½-relativo *mochila-aprox*. Resumiendo, el esquema de aproximación procede de la manera siguiente:

1. Se calcula A = *mochila-aprox*(w, v, W). Si $\varepsilon < 2n/A$, ejecutar el algoritmo de programación dinámica modificado aplicado al caso original para hallar una solución óptima exacta, empleando $M = 2A$ como cota superior de la solución óptima. En caso contrario, hacer $k = \lfloor \varepsilon A / n \rfloor \geq 2$, que ciertamente no es mayor que $\varepsilon\, opt/n$, tal como se necesita, y realizar los pasos siguientes.
2. Se forma el caso modificado en el cual $v_i' \leftarrow v_i \div k$.
3. Se utiliza el algoritmo de programación dinámica modificado para hallar una selección $X' \subseteq \{1,2,\ldots,n\}$ de objetos que maximice $\sum_{i\in X'} v_i$ sometido a la restricción $\sum_{i\in X'} w_i \leq W$. Como primer paso de este algoritmo, se utiliza $M' = \lfloor 2A/k \rfloor$ como cota superior de la solución óptima para el caso modificado. Esta es una cota superior válida porque la solución del caso original está acotada inferiormente por el producto de k por la solución óptima del caso modificado, y está acotada superiormente por $2A$.
4. Se proporciona $\widetilde{opt} = \sum_{i\in X'} v_i$ como aproximación ε–relativa de la solución óptima del problema original.

Queda por ver el tiempo que se necesita para calcular esta aproximación. El primer paso requiere un tiempo que está en $O(n \log n)$ para calcular la aproximación ½-relativa. Si $\varepsilon < 2n/A$, se necesita un tiempo que está en $O(nM)$ para obtener la solución exacta mediante programación dinámica. Este tiempo está en $O(n^2/\varepsilon)$, puesto que $nM = 2nA < 4n^2/\varepsilon$. Si $\varepsilon \geq 2n/A$, el tiempo requerido por los demás pasos se analiza de la forma siguiente. El segundo paso es despreciable. El tercer paso requiere un tiempo que está en $O(nM')$, donde $M' = \lfloor 2A/k \rfloor$ es la cota superior de la solución óptima del caso modificado que se emplea en el algoritmo de programación dinámica. Este tiempo también está en $O(n^2/\varepsilon)$, puesto que $k > \frac{\varepsilon A}{n} - 1 \geq \varepsilon A / 2n$.

El último paso es despreciable. En conclusión, esta aproximación requiere un tiempo que está en $O(n^2/\varepsilon)$, que sin duda es completamente polinómico. Se conoce otro esquema que puede hallar una solución aproximada ε–relativa del problema de la mochila en un tiempo que está en $O(n \log n + n/\varepsilon^2)$; hay otra que requiere un tiempo que está en $O\left(n \log \frac{1}{\varepsilon} + 1/\varepsilon^4\right)$.

13.6 PROBLEMAS

Problema 13.1. Dar un algoritmo eficiente para determinar si se puede colorear un grafo con sólo dos colores, y en tal caso la forma de hacerlo.

Problema 13.2. En la Sección 13.1.1, mientras se demostraba que la heurística voraz siempre puede hallar una solución óptima, comentábamos que «cuando se aplica el color 2, algunos nodos que tuvieran este color en la solución original pueden haberse pintado ya con el color 1. Sin embargo, es seguro que habrá uno o más nodos que tuvieran el color 2 en la solución original, y que no se habrán pintado». Justificar este comentario.

Problema 13.3. Demostrar que todo grafo *planar* (que se pueda colorear en una hoja de papel de tal manera que no se cruce ninguno de las aristas) se puede colorear empleando como máximo cuatro colores.

Problema 13.4. Demostrar que la heurística voraz de la sección 13.1.2 puede ser arbitrariamente mala: como función del parámetro $\alpha > 1$, construir un caso explícito del problema del viajante en el cual la heurística encuentre un recorrido que sea al menos α veces más largo que el óptimo.

Problema 13.5. Diseñar e implementar un algoritmo para determinar si una matriz de distancias $n \times n$ tiene o no la propiedad métrica. ¿Cuánto tiempo requiere su algoritmo?

Problema 13.6. Considere el grafo completo no dirigido de 8 nodos y la siguiente matriz de distancias.

Desde Hacia:	2	3	4	5	6	7	8
1	41	19	99	83	108	120	140
2		35	88	96	121	137	151
3			80	70	95	108	127
4				53	66	87	86
5					26	42	57
6						22	34
7							27

Esta matriz de distancias posee la propiedad métrica. Utilizar el algoritmo de la Sección 13.1.2 para obtener una solución aproximada del problema del viajante para este grafo. Se puede hacer sin una computadora. Si tiene a su disposición una máquina, utilice una búsqueda exhaustiva para obtener la solución exacta.

Problema 13.7. El algoritmo aproximado de la Sección 13.1.2 comienza por hallar un árbol de expansión mínimo del grafo. ¿Existe una buena razón para preferir el algoritmo de Kruskal o del de Prim en este contexto?

Problema 13.8. Para ilustrar el algoritmo aproximado de la Sección 13.1.2, después de hallar un árbol de expansión mínimo del grafo seleccionamos arbitrariamente el nodo

uno como raíz del árbol. Sin embargo, cualquier otro nodo sería igualmente válido. Empleando el ejemplo de la Sección 13.1.2, explorar lo que sucede si seleccionamos otro nodo como raíz. ¿Supone el orden de izquierda a derecha de las ramas alguna diferencia a efectos de la aproximación hallada?

Problema 13.9. En algunos casos, es posible hallar un recorrido óptimo más corto para el viajante si se permite que pase más de una vez por cada ciudad. Dar un ejemplo explícito que ilustre esto. Por otra parte, demuestre que si la matriz de distancias tiene la propiedad métrica, entonces nunca resulta ventajoso pasar más de una vez por la misma ciudad.

Problema 13.10. Vimos en la Sección 13.1.2 un algoritmo aproximado eficiente para el problema del viajante métrico, del cual se garantiza que encuentra una solución que dista de la óptima por un factor 2, como máximo. Dar otro algoritmo eficiente que tenga garantizado hallar una solución que diste de la óptima por un factor 3/2 como máximo.

Problema 13.11. Vimos en la Sección 13.2.2 un algoritmo aproximado, *mochila-aprox*, para el problema de la mochila, que proporciona una respuesta de la cual se garantiza que será al menos la mitad de la carga óptima. El primer paso de este algoritmo consiste en ordenar los n objetos por orden de valores por peso. Por esta razón, el algoritmo requiere un tiempo que está en $\Omega(n \log n)$ en el caso peor. Buscar una forma de calcular la misma aproximación en un tiempo que esté en $O(n)$ mediante el uso del algoritmo de búsqueda de la mediana en tiempo lineal que se da en la Sección 7.5.

Problema 13.12. Vimos en la Sección 13.2.2 que el algoritmo voraz de la Sección 8.4 puede ser arbitrariamente malo al resolver el problema de la mochila. Demostrar que esto no es así cuando se considera la variante del problema de la mochila estudiado en las Secciones 9.6.1 y 9.7.2: se garantiza que el algoritmo voraz proporcionará una solución que diste de la óptima en un factor 2 si se pueden utilizar tantas copias como se desee de cada objeto disponible.

Problema 13.13. Se nos dan 9 objetos cuyos pesos son respectivamente de 2, 2, 2, 3, 3, 4, 5, 6 y 9, y un cierto número de cajas de capacidad 12. Utilizar los algoritmos aproximados de la Sección 13.2.3 para estimar (a) cuál es el mayor número de objetos que se pueden almacenar en 2 cajas, y (b) cuántas cajas se necesitan para guardar los 9 objetos. Busque las soluciones óptimas y compárelas con sus respuestas aproximadas.

Problema 13.14. ¿Es significativo considerar las aproximaciones ε–relativas para problemas de maximización cuando $\varepsilon \geq 1$?

Problema 13.15. Para toda constante c, demostrar que es tan difícil hallar una solución aproximada c-absoluta para el problema de la mochila como hallar una solución exacta.

Problema 13.16. Demostrar que hallar una solución aproximada c-absoluta para el problema de la camarilla máxima es de dificultad NP para toda constante positiva c. Para ello, demostrar que CLKO $\leq_1^p c - abs -$ CLKO para todo c, en donde CLKO se definió en el problema 12.45. Puede utilizar los resultados de los problemas 12.45 y 12.55.

Problema 13.17. Prosiguiendo con el problema 13.16, demostrar que la búsqueda de una solución aproximada ε–relativa para el problema de la camarilla máxima es de dificultad NP para toda constante positiva ε mayor que 1. Tenga en cuenta que esto es muy difícil.

Problema 13.18. Si no le sale el problema 13.17, demuestre que para cualesquiera constantes positivas ε y δ menores que 1, la búsqueda de una aproximación ε–relativa para el problema de la camarilla máxima es equivalente polinómicamente a hallar una aproximación δ–relativa. Simbólicamente, demostrar que $\varepsilon - rel - \text{CLKO} \leq_T^p \delta - rel - \text{CLKO}$.

Problema 13.19. Prosiguiendo con el problema 13.3, demostrar que es fácil calcular aproximaciones 1-absolutas así como $\frac{1}{3}$-relativas del problema consistente en colorear un grafo planar con el mínimo número de colores. Por otra parte, demostrar que calcular unas aproximaciones mejores sería tan difícil como calcular la solución óptima.

Problema 13.20. Demuestre que hallar una solución aproximada ε–relativa del problema de llenado de cajas de la Sección 13.2.3 es de dificultad NP para cualquier $\varepsilon < \frac{1}{2}$. Puede utilizar sin demostración el hecho de que PARTICIÓN es NP-completo; véase el problema 12.56.

Problema 13.21. Demostrar que es tan difícil hallar una solución 1-relativa del número cromático de los grafos generales como hallar la solución exacta.

Problema 13.22. Prosiguiendo con el problema 13.21, demostrar que es tan difícil hallar una aproximación ε–relativa del número cromático de los grafos generales como hallar la solución exacta para todo $\varepsilon > 0$.

Problema 13.23. Demostrar que el algoritmo *MAX-CORTAR-aprox* de la demostración del teorema 13.4.1, del cual se garantiza que hallará una aproximación $\frac{1}{3}$-relativa del problema MAX-CORTAR, puede producir unas aproximaciones arbitrariamente malas del problema MIN-RACIMO. Para ello, mostrar la forma de asignar unos costes enteros positivos a las aristas del grafo completo en cuatro nodos como funciones de un $\alpha > 1$ arbitrario, de tal manera que la solución aproximada que halle el algoritmo para el problema MIN-RACIMO sea al menos α veces mayor que la óptima.

Problema 13.24. Los problemas de corte máximo y racimo mínimo se pueden generalizar en la forma evidente hasta el caso en que deseamos crear k racimos para cualquier constante $k \geq 2$. Esto da lugar a los problemas k-MAX-CORTAR y k-MIN-RACIMO. Aunque hemos presentado estos problemas con $k = 3$ para hacer más sencilla la demostración del Teorema 13.4.2, es más frecuente definir MAX-CORTAR y MIN-RACIMO con $k = 2$.

(a) Dar un algoritmo aproximado $\frac{1}{k}$-relativo eficiente para k-MAX-CORTAR para todo $k \geq 2$.
(b) Para todo $k \geq 3$ y $\varepsilon > 0$, demostrar que hallar una solución aproximada ε-relativa para k-MIN-RACIMO es tan difícil como hallar una solución exacta.

Problema 13.25. Proseguimos con el problema 13.24, para el caso concreto $k = 2$.

(a) Demostrar que 2-MAX-CORTAR (y por tanto 2-MIN-RACIMO) es de dificultad NP. *Sugerencia*: Utilice el hecho de que PARTICIÓN es NP-completo; véase el problema 12.56.
(b) Sabemos por el problema 13.24 que existe un algoritmo aproximado $\frac{1}{2}$-relativo eficiente para 2-MAX-CORTAR; buscar un algoritmo aproximado que sea mejor. En el momento de escribir estas líneas, el mejor algoritmo conocido es una aproximación 0,12144-relativa.

(c) Por otra parte, demostrar la existencia de un ε positivo tal que sea de dificultad NP hallar una solución aproximada ε–relativa para 2-MAX-CORTAR.

(d) Demostrar que hallar una solución aproximada ε–relativa de 2-MIN-RACIMO para todo ε positivo es de dificultad NP. (La demostración del teorema 13.4.1 falla en el caso de dos racimos, porque es sencillo decidir si un grafo se puede colorear con dos colores.)

Problema 13.26. Dar una demostración elemental de que no puede existir un esquema de aproximación completamente polinómico para el problema de la camarilla máxima, a no ser que sea P = NP. A la vista del problema 13.17, esto es evidente, pero no se conocen demostraciones sencillas ni siquiera de la existencia de un $\varepsilon < 1$ positivo para el cual la búsqueda de una solución aproximada ε–relativa del problema de la camarilla máxima sea de dificultad NP. Se puede utiliza el hecho de que el problema de la camarilla máxima es de dificultad NP.

Problema 13.27. Considere el caso siguiente del problema de la mochila. Hay cuatro objetos, cuyos pesos son respectivamente 2, 5, 6 y 7 unidades, y cuyos valores son 1, 3, 4 y 5. Podemos transportar un máximo de 11 unidades de peso:

(a) Aplicar el algoritmo aproximado voraz $\frac{1}{2}$-relativo de la Sección 13.2.2 a este caso. Deducir una cota superior del valor de la solución óptima. (Hay dos formas de obtener esta cota superior: una es evidente y la otra es sutil.)

(b) Una vez que se tiene una cota superior del valor de la solución óptima, utilícela para aplicar el algoritmo de programación dinámica dado en la Sección 13.5.2 para buscar una solución óptima. Dar una tabla similar a la figura 8.4. Determinar no sólo el valor óptimo que se puede transportar, sino también la lista de objetos que habría que llevar.

Problema 13.28. En la Sección 8.4 estudiábamos un algoritmo de programación dinámica para la solución exacta del problema de la mochila, y lo aplicamos a un caso que constaba de cinco objetos, cuyos pesos son 1,2, 5, 6 y 7 unidades, y cuyos valores son 1, 6, 18, 22 y 28. Podemos transportar un máximo de 11 unidades de peso. Aplique el esquema de aproximación completamente polinómico dado en la Sección 13.5.2 al mismo caso, con $\varepsilon = 0{,}75$. Si la solución del problema 13.27 no le parece relevante, es probable que esté siguiendo un camino equivocado (En la práctica, sería poco inteligente aplicar este algoritmo con $\varepsilon = 0{,}75$ puesto que el algoritmo voraz de la Sección 13.2.2, mucho más sencillo, es una aproximación $\frac{1}{2}$-relativa, que es mucho mejor que una aproximación $\frac{3}{4}$-relativa. Desafortunadamente, el esquema de aproximación completamente polinómico sólo resulta útil en aquellos casos que son demasiado grandes para resolverlos manualmente.)

13.7 REFERENCIAS Y TEXTOS MÁS AVANZADOS

De entre las primeras referencias de algoritmos de aproximación son buenas las de *Garey y Johnson* (1976) y *Sahni y Horowitz* (1978). Se dan algoritmos aproximados para el problema del viajante métrico en *Christofides* (1976), que contiene la solución del problema 13.10. *Laporte* (1992) hace un estudio de los algoritmos exactos y aproximados para el problema

general. En *Sahni* (1975) se dan los primero algoritmos aproximados para el problema de la mochila. *Johnson* (1973) demuestra las cotas de los algoritmos aproximados de llenado de cajas que se describen en la Sección 13.2.3. Los resultados fuertes de no aproximación para el problema de la camarilla máxima y para el problema de búsqueda del número cromático de un grafo provienen de *Bellare* y *Sudan* (1994). Se basaron en trabajos anteriores de *Arora, Lund, Motwani, Sudan* y *Szegedy* (1992), que da la solución del problema 13.17. El esquema de aproximación que describimos para el problema de la mochila procede de *Ibarra* y *Kim* (1975). Otros esquemas de aproximación para el problema de la mochila se dan en *Lawler* (1979), lugar en que se hallará la solución del problema 13.11. El problema 13.21 proviene de *Johnson* (1974). El problema 13.25 b es de *Goemans* y *Williamson* (1994).

Referencias

ABRAMSON, Bruce y Mordechai M. YUNG (1989), "Construction through decomposition: A divide-and-conquer algorithm for the N-queens problem", *Journal of Parallel and Distributed Computing*, vol. 6, n.º 3, pp. 649-662.

ACKERMANN, Wilhelm (1928), "Zum Hilbertschen Aufbau der reellen Zahlen", *Mathematische Annalen*, vol. 99, pp. 118-133.

ADEL'SON-VEL'SKIĬ, Georgiĭ M. y Evgeniĭ M. LANDIS (1962), "An algorithm for the organization of information" (en Ruso), *Doklady Akademii Nauk SSSR*, vol. 146, pp. 263-266.

ADLEMAN, Leonard M. y Ming-Deh A. HUANG (1992), *Primality Testing and Abelian Varieties Over Finite Fields*, Lecture Notes in Mathematics, vol. 1512, Springer-Verlag.

ADLEMAN, Leonard M. y Kenneth MANDERS (1977), "Reducibility, randomness, and intractability", *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pp. 151-163.

ADLEMAN, Leonard M., Carl POMERANCE y Robert S. RUMELY (1983), "On distinguishing prime numbers from composite numbers", *Annals of Mathernatics*, vol. 117, pp. 173-206.

AHMES (1700bc), *Directions for Obtaining the Knowledge of All Dark Things*, Papiro egipcio "Rhind" que se conserva en el British Museum.

AHO, Alfred V., John E. HOPCROFT y Jeffrey D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley.

AHO, Alfred V., John E. HOPCROFT y Jeffrey D. ULLMAN (1976), "On finding lowest common ancestors in trees", *SIAM Journal on Computing*, vol. 5, n.º 1, pp. 115-132.

AHO, Alfred V., John E. HOPCROFT y Jeffrey D. ULLMAN (1983), *Data Structures and Algorithms* Addison-Wesley.

AKL, Selim G. (1985), *Parallel Sorting Algorithms*, Academic Press.

AKL, Selim G. (1989), *The Design and Analysis of Parallel Algorithms*, Prentice-Hall.

ALFORD, W. R., A. GRANVILLE y Carl POMERANCE (1994), "There are infinitely many Carmichael numbers", *Annals of Mathematics*, vol. 139, pp. 703-722.

ALLISON, L., C. N. YEE y M. MCGAUGHEY (1989), "Three-dimensional queens problems" Technical Report n.º 89/130, Department of Computer Science, Monash University, Australia.

ARLAZAROV, V. L., E. A. DINIC, M. A. KRONROD y I. A. FARADŽEV (1970), "On economical construction of the transitive closure of a directed graph" (en Ruso), *Doklady Akademii Nauk SSSR* vol. 194, pp. 487-488.

ARORA, Sanjeev, Carsten LUND, Rajeev MOTWANI, Madhu SUDAN y Mario SZEGEDY (1992), "Proof verification and hardness of approximation problems", *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pp. 14-23.

BAASE, Sara (1978), *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley; 2.ª Edición, 1987.

BABAI, László (1979), "Monte Carlo algorithms in graph isomorphism techniques", Research Report n.º 79-10, Departement de mathématiques et de statistique, Université de Montréal.

BABAI, László y Shlomo MORAN (1988), "Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes", *Journal of Computer and System Sciences*, vol. 36, pp. 254-276.

BACH, Eric, Gary MILLER y Jeffrey SHALLIT (1986), "Sums of divisors, perfect numbers and factoring", *SIAM Journal on Computing*, vol. 15, n.º 4, pp. 1143-1154.

BACHMANN, Paul G. H. (1894), *Zahlentheorie; Volume 2: Die Analytische Zahlentheorie*, B. G. Teubner.

BALL, Walter W. R. (1967), *Mathematical Recreations and Essays*, 11.ª Edición, Macmillan & Co.

BATCHER, Kenneth E. (1968), "Sorting networks and their applications", *AFIPS Conference Proceedings*, vol. 32, Spring Joint Computer Conference 1968, pp. 307-314.

BEAUCHEMIN, Pierre, Gilles BRASSARD, Claude CRÉPEAU, Claude GOUTIER y Carl POMERANCE (1988), "The generation of random numbers that are probably prime", *Journal of Cryptology*, vol. 1, n.º 1, pp. 53-64.

BELLARE, Mihir y Shafi GOLDWASSER (1994), "The complexity of decision versus search", *SIAM Journal on Computing*, vol. 23, n.º 1, pp. 97-119.

BELLARE, Mihir y Madhu SUDAN (1994), "Improved non-approximability results", *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pp. 184-193.

BELLMAN, Richard E. (1957), *Dynamic Programming*, Princeton University Press.

BELLMAN, Richard E. y Stuart E. DREYFUS (1962), *Applied Dynamic Programming*, Princeton University Press.

BELLMORE M. y George NEMHAUSER (1968), The traveling salesman problem: A survey, *Operations Research*, vol. 16, n.º 3, pp. 538-558.

BENIOFF, Paul (1982), "Quantum Hamiltonian models of Turing machines", *Journal of Statistical Physics*, vol. 29, pp. 515-546.

BENNETT, Charles H., Gilles BRASSARD y Artur K. EKERT (1992), "Quantum Cryptography", *Scientific American*, vol. 267, n.º 4, pp. 50-57.

BENTLEY, Jon L. (1980), "Multidimensional divide-and-conquer", *Communications of the ACM*, vol. 23, pp. 214-229.

BENTLEY, Jon L. (1984), "Programming pearls: Algorithm design techniques", *Communications of the ACM*, vol. 27, n.º 9, pp. 865-871.

BENTLEY, Jon L., Dorothea HAKEN y James B. SAXE (1980), "A general method for solving divide-and-conquer recurrences", *ACM Sigact News*, vol. 12, n.º 3, pp. 36-44.

BENTLEY, Jon L., Donald F. STANAT y J. Michael STEELE (1981), "Analysis of a randomized data structure for representing ordered sets", *Proceedings of the 19th Annual Allerton Conference on Communication, Control, and Computing*, pp. 364-372.

BERGE, Claude (1958), *Théorie des graphes et ses applications*, Dunod; 2.ª Edición, 1967. Traducido como *The Theory of Graphs and Its Applications*, Methuen, 1962.

BERGE, Claude (1970), *Graphes et hypergraphes*, Dunod. Traducido como *Graphs and Hypergraphs*, North Holland, 1973.

BERLEKAMP, Elwyn R., John H. CONWAY y Richard K. GUY (1982), *Winning Ways for Your Mathematical Plays; Volume 1: Games in General*, Academic Press.

BERLINER, Hans J. (1980), "Backgammon computer program beats world champion", *Artificial Intelligence*, vol. 14, pp. 205-220.

BERNSTEIN, Ethan y Umesh V. VAZIRANI (1993), "Quantum complexity theory", *Proceedings of 25th Annual ACM Symposium on Theory of Computing*, pp. 11-20.

BERTHIAUME, André y Gilles BRASSARD (1995), "Oracle quantum computing", *Journal of Modern Optics*, vol. 41, n.º 12, pp. 2521-2535.

BISHOP, Errett (1972), "Aspects of constructivism", *10th Holiday Mathematics Symposium*, New Mexico State University, Las Cruces.

BITTON, Dina, David J. DEWITT, David K. HSIAO y Jaishankar MENON (1984), "A taxonomy of parallel sorting", *Computing Surveys*, vol. 16, n.º 3, pp. 287-318.

BLUM, Leonore, Manuel BLUM y Mike SHUB (1986), "A simple unpredictable pseudorandom number generator", *SIAM Journal on Computing*, vol. 15, n.º 2, pp. 364-383.

BLUM, Manuel (1967), "A machine independent theory of the complexity of recursive functions", *Journal of the ACM*, vol. 14, n.º 2, pp. 322-336.

BLUM, Manuel, Robert W. FLOYD, Vaughan R. PRATT, Ronald L. RIVEST y Robert E. TARJAN (1972), "Time bounds for selection", *Journal of Computer and System Sciences*, vol. 7, n.º 4, pp. 448-461.

BLUM, Manuel y Silvio MICALI (1984), "How to generate cryptographically strong sequences of pseudo-random bits", *SIAM Journal on Computing*, vol. 13, n.º 4, pp. 850-864.

BORODIN, Allan B. y J. Ian MUNRO (1975), *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier.

BORŮVKA, Otokar (1926), "O jistém problému minimálnim", *Práce Morasské Přirodověd Spolecnosti*, vol. 3, pp. 37-58.

BRASSARD, Gilles (1979), "A note on the complexity of cryptography", *IEEE Transactions on Information Theory*, vol. IT-25, n.º 2, pp. 232-233.

BRASSARD, Gilles (1985), "Crusade for a better notation", *ACM Sigact News*, vol. 17, n.º 1, pp. 60-64.

BRASSARD, Gilles (1988), *Modern Cryptology: A Tutorial*, Lecture Notes in Computer Science, vol. 325, Springer-Verlag.

BRASSARD, Gilles (1994), "Cryptology column—Quantum computing: The end of classical cryptography?", *ACM Sigact News*, vol. 25, n.º 4, pp. 15-21.

BRASSARD, Gilles y Paul BRATLEY (1988), *Algorithmics: Theory and Practice*, Prentice-Hall.

BRASSARD, Gilles, Sophie MONET y Daniel ZUFFELLATO (1986), "L'arithmétique des très grands entiers", *TSI: Technique et Science Informatiques*, vol. 5, n.º 2, pp. 89-102.

BRATLEY, Paul, Bennett L. FOX y Linus E. SCHRAGE (1983), *A Guide to Simulation*, Springer-Verlag; 2.ª Edición, 1987.

BRENT, Richard P. (1974), "The parallel evaluation of general arithmetic expressions", *Journal of the ACM*, vol. 21, n.º 2, pp. 201-206.

BRESSOUD, David M. (1989), *Factorization and Primality Testing*, Springer-Verlag.

BRIGHAM, E. Oran (1974), *The Fast Fourier Transform*, Prentice-Hall.

BROWN, Mark R. (1978), "Implementation and analysis of binomial queue algorithms", *SIAM Journal on Computing*, vol. 7, n.º 3, pp. 298-319.

BUNCH, James R. y John E. HOPCROFT (1974), "Triangular factorization and inversion by fast matrix multiplication", *Mathematics of Computation*, vol. 28, n.º 125, pp. 231-236.

BUNEMAN, Peter y Leon LEVY (1980), "The Towers of Hanoi problem", *Information Processing Letters*, vol. 10, n.ºs 4-5, pp. 243-244.

CALINGER, Ronald (ed.) (1982), *Classics of Mathematics*, Moore Publishing Co.

CARASSO, Claude (1971), *Analyse numérique*, Lidec.

CARLSSON, Svante (1986), *Heaps*, Tesis Doctoral, Department of Computer Science, Lund University, Sweden.

CARLSSON, Svante (1987a), "Average case results on heapsort", *BIT*, vol. 27, pp. 2-17.

CARLSSON, Svante (1987b), "The deap—A double-ended heap to implement double-ended priority queues", *Information Processing Letters*, vol. 26, n.º 1, pp. 33-36.

CARTER, J. Larry y Mark N. WEGMAN (1979), "Universal classes of hash functions", *Journal of Computer and System Sciences*, vol. 18, n.º 2, pp. 143-154.

CELIS, Pedro, Per-Åke LARSON y J. Ian MUNRO (1985), "Robin Hood hashing", *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pp. 281-288.

CHANG, Lena y James F. KORSH (1976), "Canonical coin changing and greedy solutions", *Journal of the ACM*, vol. 23, n.º 3, pp. 418-422.

CHERITON, David y Robert E. TARJAN (1976), "Finding minimum spanning trees", *SIAM Journal on Computing*, vol. 5, n.º 4, pp. 724-742.

CHIN, Francis Y., John LAM y I-Ngo CHEN (1982), "Efficient parallel algorithms for some graph problems", *Communications of the ACM*, vol. 25, n.º 9, pp. 659-665.

CHRISTOFIDES, Nicos (1975), *Graph Theory: An Algorithmic Approach*, Academic Press.

CHRISTOFIDES, Nicos (1976), "Worst-case analysis of a new heuristic for the traveling salesman problem", Research Report n.º 388, Management Sciences, Carnegie-Mellon University, Pittsburgh, PA.

COBHAM, Alan (1964), "The intrinsic computational difficulty of functions", *Proceedings of the 1964 Congress on Logic, Mathematics and the Methodology of Science*, North-Holland, pp. 24-30.

COHEN, Henri y Arjen K. LENSTRA (1987), "Implementation of a new primality test", *Mathematics of Computation*, vol. 48, n.º 177, pp. 103-121.

COLE, Richard (1988), "Parallel merge sort", *SIAM Journal on Computing*, vol. 17, n.º 4, pp. 770-785.

COOK, Steven A. (1971), "The complexity of theorem-proving procedures", *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pp. 151-158.

COOK, Steven A. y Staal O. AANDERAA (1969), "On the minimum complexity of functions", *Transactions of the American Mathematical Society*, vol. 142, pp. 291-314.

COOLEY, James W., Peter A. W. LEWIS y Peter D. WELCH (1967), "History of the fast Fourier transform", Proceedings of the IEEE, vol. 55, pp. 1675-1679.

COOLEY, James W. y John W. TUKEY (1965), "An algorithm for the machine calculation of complex Fourier series", *Mathematics of Computation*, vol. 19, n.º 90, pp. 297-301.

COPPERSMITH, Don y Shmuel WINOGRAD (1990), "Matrix multiplication via arithmetic progressions", *Journal of Symbolic Computation*, vol. 9, pp. 251-280.

CORMEN, Thomas H., Charles E. LEISERSON, y Ronald L. RIVEST (1990), *Introduction to Algorithms*, MIT Press y McGraw-Hill.

COUVREUR, Chantal y Jean-Jacques QUISQUATER (1982), "An introduction to fast generation of large prime numbers", *Philips Journal of Research*, vol. 37, pp. 231-264; errata (1983), *ibid.*, vol. 38, p. 77.

CURTISS, John H. (1956), "A theoretical comparison of the efficiencies of two classical methods and a Monte Carlo method for computing one component of the solution of a set of linear algebraic equations", del *Symposium on Monte Carlo Methods*, H. A. Meyer (ed.), Wiley, pp. 191-233.

DAMGÅRD, Ivan B., Peter LANDROCK y Carl POMERANCE (1993), "Average case error estimates for the strong probable prime test", *Mathematics of Computation*, vol. 61, n.º 203, pp. 177-194.

DANIELSON, G. C. y C. LANCZOS (1942), "Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids", *Journal of the Franklin Institute*, vol. 233, pp. 365-380, 435-452.

DE BRUIJN, Nicolaas G. (1961), *Asymptotic Methods in Analysis*, North-Holland.

DENNING, Dorothy E. R. (1983), *Cryptography and Data Security*, Addison-Wesley.

DEUTSCH, David (1985), "Quantum theory, the Church-Turing principle and the universal quantum computer", *Proceedings of the Royal Society*, London, vol. A400, pp. 97-117.

DEUTSCH, David y Richard JOZSA (1992), "Rapid solution of problems by quantum computation", *Proceedings of the Royal Society*, London, vol. A439, pp. 553-558.

DEVROYE, Luc (1986), *Non-Uniform Random Variate Generation*, Springer-Verlag.

DEWDNEY, Alexander K. (1984), "Computer recreations—Yin and yang: Recursion and iteration, the Tower of Hanoi and the Chinese rings", *Scientific American*, vol. 251, n.º 5, pp. 19-28.

DEYONG, Lewis (1977), *Playboy's Book of Backgammon*, Playboy Press.

DIFFE, Whitfield y Martin E. HELLMAN (1976), "New directions in cryptography", *IEEE Transactions on Information Theory*, vol. IT-22, n.º 6, pp. 644-654.

DIJKSTRA, Edsger W. (1959), "A note on two problems in connexion with graphs", *Numerische Mathematik*, vol. 1, pp. 269-271.

DIXON, John D. (1981), "Asymptotically fast factorization of integers", *Mathematics of Computation*, vol. 36, n.º 153, pp. 255-260.

DROMEY, R. G. (1982), *How to Solve It by Computer*, Prentice-Hall.

DUNCAN, Ralph (1990), "A survey of parallel computer architectures", *Computer*, vol. 23, n.º 2, pp. 5-16.

EDMONDS, Jack (1965), "Paths, trees, and flowers", *Canadian Journal of Mathematics*, vol. 17, n.º 3, pp. 449-467.

EDMONDS, Jack (1971), "Matroids and the greedy algorithm", *Mathematical Programming*, vol. 1, pp. 127-136.

ELKIES, Noam D. (1988), "On $A^4 + B^4 + C^4 = D^4$", *Mathematics of Computation*, vol. 51, n.º 184, pp. 825-835.

ERDÖS, Paul y Carl POMERANCE (1986), "On the number of false witnesses for a composite number", *Mathematics of Computation*, vol. 46, n.º 173, pp. 259-279.

EVEN, Shimon (1980), *Graph Algorithms*, Computer Science Press.

EVES, Howard (1983), *An Introduction to the History of Mathematics*, 5.ª Edición, Saunders College Publishing.

FEYNMAN, Richard (1982), "Simulating physics with computers", *International Journal of Theoretical Physics*, vol. 21, n.ºs 6/7, pp. 467-488.

FEYNMAN, Richard (1986), "Quantum mechanical computers", *Foundations of Physics*, vol. 16, n.º 6, pp. 507-531; originally appeared in *Optics News*, February 1985.

FISCHER, Michael J. y Albert R. MEYER (1971), "Boolean matrix multiplication and transitive closure", *Proceedings of the 12th Annual IEEE Symposium on Switching and Automata Theory*, pp. 129-131.

FLAJOLET, Philippe (1985), "Approximate counting: A detailed analysis", *BIT*, vol. 25, pp. 113-134.

FLAJOLET, Philippe y G. Nigel MARTIN (1985), "Probabilistic counting algorithms for data base applications", *Journal of Computer and System Sciences*, vol. 31, n.º 2, pp. 182-209.

FLOYD, Robert W. (1962), "Algorithm 97: Shortest path", *Communications of the ACM*, vol. 5, n.º 6, p. 345.

FOX, Bennett L. (1986), "Algorithm 647: Implementation and relative efficiency of quasirandom sequence generators", *ACM Transactions on Mathematical Software*, vol. 12, n.º 4, pp. 362-376.

FREDMAN, Michael L. (1976), "New bounds on the complexity of the shortest path problem", *SIAM Journal on Computing*, vol. 5, n.º 1, pp. 83-89.

FREDMAN, Michael L. y Robert E. TARJAN (1987), "Fibonacci heaps and their use in improved network optimization algorithms", *Journal of the ACM*, vol. 34, n.º 3, pp. 596-615.

FREDMAN, Michael L. y Dan E. WILLARD (1990), "BLASTING through the information theoretic barrier with FUSION TREES", *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pp. 1-7.

FREIVALDS, Rūsiņš (1977), "Probabilistic machines can use less running time", *Proceedings of Information Processing '77*, pp. 839-842.

FREIVALDS, Rūsiņš (1979), "Fast probabilistic algorithms", *Proceedings of the 8th Symposium on the Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, vol. 74, Springer-Verlag.

FURMAN, M. E. (1970), "Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph" (en Ruso), *Doklady Akademii Nauk SSSR*, vol. 194 p. 524.

GALIL, Zvi y Giuseppe F. ITALIANO (1991), "Data structures and algorithms for disjoint set union problems", *Computing Surveys*, vol. 23, n.º 3, pp. 319-344.

GARDNER, Martin (1959), *The Scientific American Book of Mathematical Puzzles and Diversions*, Simon and Schuster.

GARDNER, Martin (1977), "Mathematical games: A new kind of cipher that would take millions of years to break", *Scientific American*, vol. 237, n.º 2, pp. 120-124.

GARDNER, Martin y Charles H. BENNETT (1979), "Mathematical games: The random number omega bids fair to hold the mysteries of the universe", *Scientific American*, vol. 241, n.º 5, pp. 20-34.

GAREY, Michael R., Ronald L. GRAHAM y David S. JOHNSON (1977), "The complexity of computing Steiner minimal trees", *SIAM Journal on Applied Mathematics*, vol. 32, pp. 835-859.

G AREY, Michael R. y David S. JOHNSON (1976), "Approximation algorithms for combinatorial problems: An annotated bibliography", de *Algorithms and Complexity: Recent Results and New Directions*, J. F. Traub (ed.), Academic Press, pp. 41-52.

GAREY, Michael R. y David S. JOHNSON (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman.

GIBBONS, Alan y Wojciech RYTTER (1988), *Efficient Parallel Algorithms*, Cambridge University Press.

GILBERT, Edgard N. y Edward F. MOORE (1959), "Variable length encodings", *Bell System Technical Journal*, vol. 38, n.º 4, pp. 933-968.

GILL, John (1977), "Computational complexity of probabilistic Turing machines", *SIAM Journal on Computing*, vol. 6, pp. 675-695.

GODBOLE, Sadashiva S. (1973), "On efficient computation of matrix chain products", *IEEE Transactions on Computers*, vol. C-22, n.º 9, pp. 864-866.

GOEMANS, Michel X. y David P. WILLIAMSON (1994), ".878-Approximation algorithms for MAX CUT and MAX 2SAT", *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pp. 422-431.

GOLDREICH, Oded, Silvio MICALI y Avi WIGDERSON (1991), "Proofs that yield nothing but their validity, or all languages in NP have zero-knowledge proof systems", *Journal of the ACM*, vol. 38, pp. 691 -729.

GOLDWASSER, Shafi y Joe KILIAN (1986), "Almost all primes can be quickly certified", *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pp. 316-329.

GOLDWASSER, Shafi, Silvio MICALI y Charles RACKOFF (1989), "The knowledge complexity of interactive proof-systems", *SIAM Journal on Computing*, vol. 18, pp. 186-208.

GOLOMB, Solomon W. y Leonard D. BAUMERT (1965), "Backtrack programming", *Journal of the ACM*, vol. 12, n.º 4, pp. 516-524.

GONDRAN, Michel y Michel MINOUX (1979), *Graphes et algorithmes*, Eyrolles. Traducido como *Graphs and Algorithms*, Wiley, 1984.

GONNET, Gaston H. y Ricardo BAEZA-YATES (1984), *Handbook of Algorithms and Data Structures*. Addison-Wesley; 2.ª Edición, 1991.

GONNET, Gaston H. y J. Ian MUNRO (1986), "Heaps on heaps", *SIAM Journal on Computing*, vol. 15, n.º 4, pp. 964-971.

GOOD, Irving J. (1968), "A five-year plan for automatic chess", *de Machine Intelligence* 2, E. Dale y D. Michie (eds), American Elsevier.

GRAHAM, Ronald L. y Pavol HELL (1985), "On the history of the minimum spanning tree problem", *Annals of the History of Computing*, vol. 7, n.º 1, pp. 43-57.

GREENE, Daniel H. y Donald E. KNUTH (1981), *Mathematics for the Analysis of Algorithms*, Birkhauser.

GRIES, David (1981), *The Science of Programming*, Springer-Verlag.

GRIES, David y Gary LEVIN (1980), "Computing Fibonacci numbers (and similarly defined functions) in log time", *Information Processing Letters*, vol. 11, n.º 2, pp. 68-69.

GUIBAS, Leonidas J. y Robert SEDGEWICK (1978), "A dichromatic framework for balanced trees", *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pp. 8-21.

GUY, Richard K. (1981), *Unsolved Problems in Number Theory*, Springer-Verlag.

HALL, A. (1873), "On an experimental determination of π", *Messenger of Mathematics*, vol. 2, pp. 113-114.

HAMMERSLEY, John M. y David C. HANDSCOMB (1965), *Monte Carlo Methods*; reprinted by Chapman and Hall, 1979.

HARDY, Godfrey H. y Edward M. WRIGHT (1938), *An Introduction to the Theory of Numbers*, Oxford University Press; 5.ª Edición, 1979.

HAREL, David (1987), *Algorithmics: The Spirit of Computing*, Addison-Wesley; 2.ª Edición, 1992.

HEATH, Sir Thomas L. (1926), *The Thirteen Books of Euclid's Elements*, 3 volumes, 2.ª Edición, Cambridge University Press; reprinted by Dover Publications, 1956.

HELD, M. y Richard KARP (1962), "A dynamic programming approach to sequencing problems", *SIAM Journal on Applied Mathematics*, vol. 10, n.º 1, pp. 196-210.

HELLMAN, Martin E. (1980), "The mathematics of public-key cryptography", *Scientific American*, vol. 241, n.º 2, pp. 146-157.

HILLIER, Frederick S. y Gerald J. LIEBERMAN (1967), *Introduction to Operations Research*. Holden-Day.

HIRSCHBERG, D. S., Ashok K. CHANDRA y D. V. SARWATE (1979), "Computing connected components on parallel computers", *Communications of the ACM*, vol. 22, n.º 8, pp. 461-464.

HOARE, Charles A. R. (1962), "Quicksort", *Computer Journal*, vol. 5, n.º 1, pp. 10-15.

HOPCROFT, John E. y Richard KARP (1971), "An algorithm for testing the equivalence of finite automata", Technical Report TR-71-114, Department of Computer Science, Cornell University, Ithaca, NY.

HOPCROFT, John E. y Leslie R. KERR (1971), "On minimizing the number of multiplications necessary for matrix multiplication", *SIAM Journal on Applied Mathematics*, vol. 20, n.º 1, pp. 30-36.

HOPCROFT, John E. y Robert E. TARJAN (1973). Efficient algorithms for graph manipulation", *Communications of the ACM*, vol. 16, n.º 6, pp. 372-378.

HOPCROFT, John E. y Robert E. TARJAN (1974). Efficient planarity testing", *Journal of the ACM*, vol. 21, n.º 4, pp. 549-568.

HOPCROFT, John E. y Jeffrey D. ULLMAN (1973), "Set merging algorithms", *SIAM Journal on Computing*, vol. 2, n.º 4, pp. 294-303.

HOPCROFT, John E . y Jeffrey D. ULLMAN (1979), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley.

HOROWITZ, Ellis y Sartaj SAHNI (1976), *Fundamentals of Data Structures*, Computer Science Press.

HOROWITZ, Ellis y Sartaj SAHNI (1978), *Fundamentals of Computer Algorithms*, Computer Science Press.

HU, Te Chiang (1981), *Combinatorial Algorithms*, Addison-Wesley.

HU, Te Chiang y M. T. SHING (1982), "Computations of matrix chain products", Part I, *SIAM Journal on Computing*, vol. 11, n.º 2, pp. 362-373.

Hu, Te Chiang y M. T. SHING (1984), "Computations of matrix chain products", Part II, *SIAM Journal on Computing*, vol. 13, n.º 2, pp. 228-251.

IBARRA, Oscar H. y Chul E. KIM (1975), "Fast approximation algorithms for the knapsack and sum of subset problems", *Journal of the ACM*, vol. 22, n.º 4, pp. 463-468.

IRVING, Robert W. (1984), "Permutation backtracking in lexicographic order", *The Computer Journal*, vol. 27, n.º 4, pp. 373-375.

JA'JA', Joseph (1992), *An Introduction to Parallel Algorithms*, Addison-Wesley.

JANKO, Wolfgang (1976), "A list insertion sort for keys with arbitrary key distribution", *ACM Transactions on Mathematical Software*, vol. 2, n.º 2, pp. 143-153.

JARNÍK V. (1930), "O jistém problému minimálnim", *Práce Morasské Přírodověd Spolecnosti*, vol. 6, pp. 57-63.

JENSEN, Kathleen y Niklaus WIRTH (1985), *Pascal User Manual and Report*, 3.ª Edición revised by Andrew B. Mickel y James E Miner, Springer-Verlag.

JOHNSON, David S. (1973), *Near-Optimal Bin-Packing Algorithms*, Tesis Doctoral, Massachusetts Institute of Technology, MIT Report MAC TR-109.

JOHNSON, David S. (1974), "Approximation algorithms for combinatorial problems", *Journal of Computer and System Sciences*, vol. 9, pp. 256-289.

JOHNSON, David S. (1990), "A catalog of complexity classes", de van Leeuwen (1990), pp. 67-161.

JOHNSON, Donald B. (1975), "Priority queues with update and finding minimum spanning trees", *Information Processing Letters*, vol. 4, n.º 3, pp. 53-57.

JOHNSON, Donald B. (1977), "Efficient algorithms for shortest paths in sparse networks", *Journal of the ACM*, vol. 24, n.º 1, pp. 1-13.

KAHN, David (1966), "Modern cryptology", *Scientific American*, vol. 215, n.º 1, pp. 38-46.

KAHN, David (1967), *The Codebreakers: The Story of Secret Writing*, Macmillan.

KALISKI, Burt S., Ronald L. RIVEST y Alan T. SHERMAN (1988), "Is the Data Encryption Standard a group? (Results of cycling experiments on DES)", *Journal of Cryptology*, vol. 1, n.º 1, pp. 3-36.

KARATSUBA, Anatoliĭ A. y Y. OFMAN (1962), "Multiplication of multidigit numbers on automata" (en Ruso), *Doklady Akademii Nauk SSSR*, vol. 145, pp. 293-294.

KARP, Richard (1972), "Reducibility among combinatorial problems", de *Complexity of Computer Computations*, R. E. Miller y J. W. Thatcher (eds), Plenum Press, pp. 85-104.

KASIMI, T. (1965), "An efficient recognition and syntax algorithm for context-free languages", Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

KIM, SU Hee y Carl POMERANCE (1989), "The probability that a random probable prime is composite", *Mathematics of Computation*, vol. 53, n.º 188, pp. 721-741.

KINGSTON, Jeffrey H. (1990), *Algorithms and Data Structures: Design, Correctness, Analysis*, Addison-Wesley.

KIRKERUD, Bjørn (1989), *Object-Oriented Programming with Simula*, Addison-Wesley.

KLEENE, Stephen C. (1956), "Representation of events in nerve nets and finite automata", de *Automata Studies*, C. E. Shannon y J. McCarthy (eds), Princeton University Press, pp. 3-40.

KLINE, Morris (1972), *Mathematical Thoughts from Ancient to Modern Times*, Oxford University Press.

KNUTH, Donald E. (1968), *The Art of Computer Programming; Volume 1: Fundamental Algorithms*, Addison-Wesley; 2.ª Edición, 1973.

KNUTH, Donald E. (1969), *The Art of Computer Programming; Volume 2: Seminumerical Algorithms*, Addison-Wesley; 2.ª Edición, 1981.

KNUTH, Donald E. (1971), "Optimal binary search trees", *Acta Informatica*, vol. 1, pp. 14-25.

KNUTH, Donald E. (1973), *The Art of Computer Programming; Volume 3: Sorting and Searching*, Addison-Wesley.

KNUTH, Donald E. (1975), "Estimating the efficiency of backtrack programs", *Mathematics of Computation*, vol. 29, pp. 121 -136.

KNUTH, Donald E. (1976), "Big omicron and big omega and big theta", *ACM Sigact News*, vol. 8, n.º 2, pp. 18 24.

KNUTH, Donald E. (1977), "Algorithms", *Scientific American*, vol. 236, n.º 4, pp. 63-80.

KOBLITZ, Neal (1987), *A Course in Number Theory and Cryptography*, Springer-Verlag.

KOZEN, Dexter C. (1992), *The Design and Analysis of Algorithms*, Springer-Verlag.

KRAITCHIK, Maurice B. (1926), *Theorie des nombres*, Tome II, Gauthier-Villars.

KRANAKIS, Evangelos (1986), *Primality and Cryptography*, Wiley-Teubner Series in Computer Science.

KRUSKAL, Joseph B., Jr. (1956), "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proceedings of the American Mathematical Society*, vol. 7, n.º 1, pp. 48-50.

KUČERA, Luděk. (1982), "Parallel computation and conflicts in memory access", *Information Processing Letters*, vol. 14, n.º 2, pp. 93-96.

LAKSHMIVARAHAN, S. y Sudarshan K. DHALL (1990), *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems*, McGraw-Hill.

LAPORTE, Gilbert (1992), "The traveling salesman problem: An overview of exact and approximate algorithms", *European Journal of Operational Research*, vol. 59, pp. 231-247.

LAROUSSE (1968), *Grand Larousse Encyclopédique*, Librairie Larousse, article on "Pâque"

LAURIÈRE, Jean-Louis (1979), *Eléments de programmation dynamique*, Bordas.

LAWLER, Eugene L. (1976), *Combinatorial Optimization: Networks and Matroids*, Holt. Rinehart and Winston.

LAWLER, Eugene L. (1979), "Fast approximation algorithms for knapsack problems", *Mathematics of Operations Research*, vol. 4, pp. 339-356.

LAWLER, Eugene L. y D. W. WOOD (1966), "Branch-and-bound methods: A survey", *Operations Research*, vol. 14, n.º 4, pp. 699-719.

LECARME, Olivier y Jean-Louis NEBUT (1985), *Pascal pour programmeurs*, McGraw-Hill.

LECARME, Georges L., Comte de BUFFON (1977), *Essai d'arithmetique morale*

L'ÉCUYER, Pierre (1988), "Efficient and portable combined random number generators", *Communications of the ACM*, vol. 31, n.º 6, pp. 742-749 y 774.

L'ÉCUYER, Pierre (1990), "Random numbers for simulation", *Communications of the ACM*, vol. 33, n.º 10, pp. 85-97.

LEIGHTON, F. Thomson (1992), *Introduction to Parallel Algorithms and Architectures: Arrays·Trees·Hypercubes*, Morgan Kaufmann.

LENSTRA, Arjen K., Hendrik W. LENSTRA Jr., Mark S. MANASSE y John M. POLLARD (1993), "The number field sieve", de *The Development of the Number Field Sieve*, A. K. Lenstra y H. W. Lenstra Jr. (eds), Lecture Notes in Mathematics, vol. 1554, Springer-Verlag, pp. 11-42.

LENSTRA, Arjen K. y Mark S. MANASSE (1990), "Factoring by electronic mail", de *Advances in Cryptology—Eurocrypt'89 Proceedings*, Lecture Notes in Computer Science, vol. 434, Springer-Verlag, pp. 355-371.

LENSTRA, Arjen K. y Mark S. MANASSE (1991), "Factoring with two large primes", de *Advances in Cryptology-Eurocrypt'90 Proceedings*, Lecture Notes in Computer Science, vol. 473, Springer-Verlag, pp. 72-82.

LENSTRA, Hendrik W., Jr. (1982), "Primality testing", de Lenstra y Tijdeman (1982), pp. 55-97.

LENSTRA, Hendrik W., Jr. (1987), "Factoring integers with elliptic curves", *Annals of Mathematics*, vol. 126, n.º 3, pp. 649-673.

LENSTRA, Hendrik W., Jr. y R. TIJDEMAN (eds) (1982), *Computational Methods in Number Theory*, Part I, Mathematical Centre Tracts 154, Mathematisch Centrum, Amsterdam.

LEVIN, Leonid (1973), "Universal search problems" (en Ruso), *Problemy Peredaci Informacii*, vol. 9, pp. 115-116.

LEWIS, Harry R. y Larry DENENBERG (1991), *Data Structures & Their Algorithms*, Harper Collins Publishers.

LEWIS, Harry R. y Christos H. PAPADIMITRIOU (1978), "The efficiency of algorithms", *Scientific American*, vol. 238, n.º 1, pp. 96-109.

LLOYD, Seth (1993), "A potentially realizable quantum computer", *Science*, vol. 261, 17 September, pp. 1569-1571.

LUEKER, George S. (1980), "Some techniques for solving recurrences", *Computing Surveys*, vol. 12, n.º 4, pp. 419-436.

MANBER, Udi (1989), *Introduction to Algorithms: A Creative Approach*, Addison-Wesley.

MARSH, D. (1970), "Memo functions, the graph traverser, and a simple control situation", de *Machine Intelligence 5*, B. Meltzer y D. Michie (eds), American Elsevier and Edinburgh University Press, pp. 281-300.

MAURER, Ueli M. (1995), "Fast generation of prime numbers y secure public-key cryptographic parameters", *Journal of Cryptology*, vol. 8, n.º 3.

MCCARTY, Carl P. (1978), "Queen squares", *The American Mathematical Monthly*, vol. 85, n.º 7, pp. 578-580.

MCDIARMID, Colin J. H. y Bruce A. REED (1989), "Building heaps fast", *Journal of Algorithms*, vol. 10, n.º 3, pp. 352-365.

MELHORN, Kurt (1984a), *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag.

MELHORN, Kurt (1984b), *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag.

MELHORN, Kurt (1984c), *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag.

MERKLE, Ralph C. (1978), "Secure communications over insecure channels", *Communications of the ACM*, vol. 21, pp. 294-299.

METROPOLIS, I. Nicholas y Stanislaw ULAM (1949), "The Monte Carlo method", *Journal of the American Statistical Association*, vol.44, n.º 247, pp. 335-341.

MICHIE, Donald (1968), "'Memo' functions and machine learning", *Nature*, vol. 218, pp. 19-22.

MILLER, Gary L. (1976), "Riemann's hypothesis and tests for primality", *Journal of Computer and System Sciences*, vol.13, n.º 3, pp. 300-317.

MONIER, Louis (1980), "Evaluation and comparison of two efficient probabilistic primality testing algorithms", *Theoretical Computer Science*, vol. 12, pp. 97-108.

MORET, Bernard M. E. y Henry D. SHAPIRO (1991), *Algorithms from P to NP; Volume I: Design & Efficiency*, Benjamin/Cummings.

MORRIS, Robert (1978), "Counting large numbers of events in small registers", *Communications of the ACM*, vol. 21, n.º 10, pp. 840-842.

NAIK, Ashish V., Mitsunori OGIWARA y Alan L. SELMAN (1993), "P-selective sets, and reducing search to decision vs. self-reducibility", *Proceedings of the 8th Annual IEEE Conference on Structure in Complexity Theory*, pp. 52-64.

NELSON, C. Greg y Derek C. OPPEN (1980), "Fast decision procedures based on congruence closure", *Journal of the ACM*, vol. 27, pp. 356-364.

NEMHAUSER, George (1966), *Introduction to Dynamic Programming*, Wiley.

NIEVERGELT, Jurg y Klaus HINRICHS (1993), *Algorithms and Data Structures with Applications to Graphics and Geometry*, Prentice-Hall.

NILSSON, Nils J. (1971) *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill.

PAGEAU, Marie (1993), *Applications du probabilisme a l'algorithmique*, Tesina de Licenciatura, Département d'informatique et de R.O., Université de Montréal.

PAN, Viktor Y. (1980), "New fast algorithms for matrix operations", *SIAM Journal on Computing*, vol. 9, pp. 321-342.

PAPADIMITRIOU, Christos H. (1994), *Computational Complexity*, Addison-Wesley.

PAPADIMITRIOU, Christos H. y Kenneth STEIGLITZ (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall.

PERALTA, René C. (1986), "A simple and fast probabilistic algorithm for computing square roots modulo a prime number", *IEEE Transactions on Information Theory*, vol. IT-32, n.º 6, pp. 846 847.

PIPPENGER, Nicholas (1978), "Complexity theory", *Scientific American*, vol. 238, n.º 6, pp. 114-124.

PIPPENGER, Nicholas (1979), "On simultaneous resource bounds", *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pp. 307-311.

POLLARD, John M. (1971), "The fast Fourier transform in a finite field", *Mathematics of Computation*, vol. 25, n.º 114, pp. 365-374.

POLLARD, John M. (1975), "A Monte Carlo method of factorization", *BIT*, vol. 15, pp. 331-334.

PÓLYA, György (1945), *How to Solve It: A New Aspect of Mathematical Method*, Princeton University Press.

PÓLYA, György (1954), *Induction and Analogy in Mathematics*, Princeton University Press.

POMERANCE, Carl (1982), "Analysis and comparison of some integer factoring algorithms", de Lenstra y Tijdeman (1982), pp. 89-139.

POMERANCE, Carl (1984), "The quadratic sieve algorithm", de *Advances in Cryptology: Proceedings of Eurocrypt 84*, Lecture Notes in Computer Science, vol. 209, Springer-Verlag, pp. 169-182.

POMERANCE, Carl (1987), "Very short primality proofs", *Mathematics of Computation*, vol. 48, n.º 177, pp. 315-322.

PRATT, Vaughan R. (1975), "Every prime has a succinct certificate", *SIAM Journal on Computing*, vol. 4, n.º 3, pp. 214-220.

PRIM, Robert C. (1957), "Shortest connection networks and some generalizations", *Bell System Technical Journal*, vol. 36, pp.1389-1401.

PURDOM, Paul W., Jr. y Cynthia A. BROWN (1985), *The Analysis of Algorithms*, Holt, Rinehart and Winston.

QUINN, Michael J. y Narsingh DEO (1984), "Parallel Graph Algorithms", *Computing Surveys*, vol. 16, n.º 3, pp. 319-348.

RABIN, Michael O. (1976), "Probabilistic Algorithms", de *Algorithms and Complexity: Recent Results and New Directions*, J. F. Traub (ed.), Academic Press, pp. 21-39.

RABIN, Michael O. (1980a), "Probabilistic algorithms in finite fields", *SIAM Journal on Computing*, vol. 9, n.º 2, pp. 273-280.

RABIN, Michael O. (1980b), "Probabilistic algorithm for primality testing", *Journal of Number Theory*, vol. 12, pp. 128-138.

RAWLINS, Gregory J. E. (1992), *Compared fo What? An Introduction to the Analysis of Algorithms*, Computer Science Press.

REIF, John H. (1993), *Synthesis of Parallel Algorithms*, Morgan Kaufmann.

REINGOLD, Edward M., Jurg NIEVERGELT y Narsingh DEO (1977), *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall.

RICE, John A. (1988), *Mathematical Statistics and Data Analysis*, Duxbury Press; 2.ª Edición, 1995.

RIVEST, Ronald L. y Robert W. FLOYD (1973), "Bounds on the expected time for median computations", de *Combinatorial Algorithms*, R. Rustin (ed.), Algorithmics Press, pp. 69-76.

RIVEST, Ronald L., Adi SHAMIR y Leonard M. ADLEMAN (1978), "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, vol. 21, n.º 2, pp. 120 126.

ROBSON, John M. (1973), "An improved algorithm for traversing binary trees without auxiliary stack", *Information Processing Letters*, vol. 2, n.º 1, pp. 12-14.

ROSEN, Kenneth H. (1991), *Discrete Mathematics and Its Applications*, 2.ª Edición, McGraw-Hill.

ROSENTHAL, Arnie y Anita GOLDNER (1977), "Smallest augmentation to biconnect a graph", *SIAM Journal on Computing*, vol. 6, n.º 1, pp. 55-66.

RUNGE, Carl D. T. y Hermann KÖNIG (1924), "Vorlesungen über numerisches Rechnen", *Die Grundlehren der Mathematischen Wissenschaften*, vol. 11, Springer-Verlag, Berlin, pp. 211-237.

SAHNI, Sartaj (1975), "Approximate algorithms for the 0/1 knapsack problem", *Journal of the ACM*, vol. 22, n.º 1, pp. 115-124.

SAHNI, Sartaj y Ellis HOROWITZ (1978), "Combinatorial problems: Reducibility and approximation", *Operations Research*, vol. 26, n.º 4, pp. 718-759.

SAVITCH, Walter J. (1970), "Relationship between nondeterministic and deterministic tape classes", *Journal of Computer and System Sciences*, vol. 4, pp. 177-192.

SCHAFFER, Russel y Robert SEDGEWICK (1993), "The analysis of heapsort", *Journal of Algorithms*, vol. 15, n.º 1, pp. 76-100.

SCHARLAU, Winfried y Hans OPOLKA (1985), *From Fermat to Minkowski: Lectures on the Theory of Numbers and Its Historical Development*, Springer-Verlag.

SCHNEIER, Bruce (1994), *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, Wiley.

SCHÖNHAGE, Arnold y Volker STRASSEN (1971), "Schnelle Multiplikation grosser Zahlen", *Computing*, vol. 7, pp. 281-292.

SCHWARTZ, Eugene S. (1964), "An optimal encoding with minimum longest code and total number of digits", *Information and Control*, vol. 7, n.º 1, pp. 37-44.

SCHWARTZ, J. (1978), "Probabilistic algorithms for verification of polynomial identities", Technical Report n.º 604, Computer Science Department, Courant Institute, New York University.

SEDGEWICK, Robert (1983), *Algorithms*, Addison-Wesley; 2.ª Edición, 1988.

SHALLIT, Jeffrey (1992), "Randomized algorithms in 'primitive' cultures, or what is the oracle complexity of a dead chicken", *ACM Sigact News*, vol. 23, n.º 4, pp. 77-80; see also *ibid.* (1993), vol. 24, n.º 1, pp. 1-2.

SHAMIR, Adi (1979), "Factoring numbers in $O(\log n)$ arithmetic steps", *Information Processing Letters*, vol. 8, n.º 1, pp. 28-31.

SHOR, Peter W. (1994), "Algorithms for quantum computation: Discrete logarithms and factoring", *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 124-134.

SIMMONS, Gustavus J. (ed.) (1992), *Contemporary Cryptology: The Science of Information Integrity*, IEEE Press.

SIMON, Daniel R. (1994), "On the power of quantum computation", *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 116-123.

SLEATOR, Daniel D. y Robert E . TARJAN (1985), "Self-adjusting binary search trees", *Journal of the ACM*, vol. 32, pp. 652-686.

SLOANE, Neil J. A. (1973), *A Handbook of Integer Sequences*, Academic Press.

SOBOL', Il'ia M. (1974), *The Monte Carlo Method*, 2.ª Edición, University of Chicago Press.

SOLOMON, Herbert (1978), *Geometric Probability*, SIAM.

SOLOVAY, Robert y Volker STRASSEN (1977), "A fast Monte-Carlo test for primality", *SIAM Journal on Computing*, vol. 6, n.º 1, pp. 84-85; erratum (1978), *ibid.*, vol. 7, n.º 1, p. 118.

STANDISH, Thomas A. (1980), *Data Structure Techniques*, Addison-Wesley.

STINSON, Douglas R. (1985), *An Introduction to the Design and Analysis of Algorithms*, The Charles Babbage Research Centre, St. Pierre, Manitoba; 2.ª Edición, 1987.

STINSON, Douglas R. (1995), *Cryptography: Theory and Practice*, CRC Press, Inc.

STOCKMEYER, Larry J. (1973), "Planar 3-colorability is polynomial complete", *ACM Sigact News*, vol. 5, n.º 3, pp. 19-25.

STOCKMEYER, Larry J. y Ashok K. CHANDRA (1979), "Intrinsically difficult problems", *Scientific American*, vol. 240, n.º 5, pp. 140-159.

STONE, Harold S. (1972), *Introduction to Computer Organization and Data Structures*, McGraw-Hill.

STRASSEN, Volker (1969), "Gaussian elimination is not optimal", *Numerische Mathematik*, vol. 13, pp. 354-356.

TARJAN, Robert E. (1972), "Depth-first search and linear graph algorithms", *SIAM Journal on Computing*, vol. 1, n.º 2, pp. 146-160.

TARJAN, Robert E. (1975), "On the efficiency of a good but not linear set merging algorithm", *Journal of the ACM*, vol. 28, n.º 3, pp. 577-593.

TARJAN, Robert E. (1981), "A unified approach to path problems", *Journal of the ACM*, vol. 28, n.º 3, pp. 577-593.

TARJAN, Robert E. (1983), Data *Structures and Network Algorithms*, SIAM.

TARJAN, Robert E. (1985), "Amortized computational complexity", *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, n.º 2, pp. 306-318.

TUCKER, Lewis W. y George G. ROBERTSON (1988), "Architecture and applications of the connection machine", *Computer*, vol. 21, n.º 8, pp. 26-38.

TURING, Alan M. (1936), "On computable numbers with an application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, vol. 2, n.º 42, pp. 230-265.

TURK, J. W. M. (1982), "Fast arithmetic operations on numbers and polynomials", de Lenstra y Tijdeman (1982), pp. 43-54.

URBANEK, Friedrich J. (1980), "An O (log n) algorithm for computing the nth element of the solution of a difference equation", *Information Processing Letters*, vol. 11, n.º 2, pp. 66-67.

VAN LEEUWEN, Jan (ed.) (1990), *Handbook of Theoretical Computer Science; Volume A: Algorithms and Complexity*, Elsevier y MIT Press.

VAZIRANI, Umesh V. (1986), *Randomness, Adversaries, and Computation*, Tesis Doctoral, Computer Science, University of California, Berkeley, CA.

VAZIRANI, Umesh V. (1987), "Efficiency considerations in using semi-random sources", *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pp. 160-168.

VERMA, Rakesh M. (1994), "A general method and a master theorem for divide-and-conquer recurrences with applications", *Journal of Algorithms*, vol. 16, pp. 67-79.

VICKERY, C. W. (1956), "Experimental determination of eigenvalues and dynamic influence coefficients for complex structures such as airplanes", de *Symposium on Monte Carlo Methods*, H. A. Meyer (ed.), Wiley, pp. 145-146.

VON NEUMANN, John (1951), "Various techniques used in connection with random digits", *Journal of Research of the National Bureau of Standards, Applied Mathematics Series*, vol. 3, pp. 36-38.

VUILLEMIN, Jean (1978), "A data structure for manipulating priority queues", *Communications of the ACM*, vol. 21, n.º 4, pp. 309-315.

WAGNER, Robert A. y Michael J. FISCHER (1974), "The string-to-string correction problem", *Journal of the ACM*, vol. 21, n.º 1, pp. 168-173.

WARSHALL, Stephen (1962), "A theorem on Boolean matrices", *Journal of the ACM*, vol. 9, n.º 1, pp. 11-12.

WARUSFEL, André (1961), *Les nombres et leurs mysteres*, Editions du Seuil.

WEGMAN, Mark N. y J. Larry CARTER (1981), "New hash functions and their use in authentication and set equality", *Journal of Computer and System Sciences*, vol. 22, n.º 3, pp. 265-279.

WILLIAMS, Hugh (1978), "Primality testing on a computer", *Ars Combinatoria*, vol. 5, pp. 127-185.

WILLIAMS, John W. J. (1964), "Algorithm 232: Heapsort", *Communications of the ACM*, vol. 7, n.º 6, pp. 347-348.

WINOGRAD, Shmuel (1980), *Arithmetic Complexity of Computations*, SIAM.

WINTER, Pavel (1987), "Steiner problem in networks: A survey", *Networks*, vol. 17, n.º 2, pp. 129-167.

WOOD, Derick (1993), *Data Structures, Algorithms, and Performance*, Addison-Wesley.

WRIGHT, J. W. (1975), "The change-making problem", *Journal of the ACM*, vol. 22, n.º 1, pp. 125-128.

YAO, Andrew C.-C. (1975), "An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees", *Information Processing Letters*, vol. 4, n.º 1, pp. 21-23.

YAO, Andrew C.-C. (1982), "Theory and applications of trapdoor functions", *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pp. 80-91.

YAO, Frances F. (1980), "Efficient dynamic programming using quadrangle inequalities", *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pp. 429-435.

YOUNGER, Daniel H. (1967), "Recognition of context-free languages in time n^3", *Information and Control*, vol. 10, n.º 2, pp. 189-208.

ZIPPEL, Richard E. (1979), *Probabilistic Algorithms for Sparse Polynomials*, Tesis Doctoral, Massachusetts Institute of Technology, Cambridge, MA.

Notas finales

NOTAS AL CAPÍTULO 2

1. El término **algorithmics** se traduce en ocasiones por *algorítmica*. Sin embargo, el único término aceptado por el Diccionario de la Real Academia Española es **algoritmia**, y ésta es la razón por la que hemos decidido utilizar este término. (N. del RT.)
2. El término inglés **instance** se suele traducir también por *caso* e *instancia*. En la jerga informática es usual emplear el término **instancia.** (N. del RT.)
3. Aunque el término *invariancia* no está aceptado como tal por el DRAE (Diccionario de la Lengua de la Real Academia Española) sí están aceptados otros términos relacionados tales como *invariación* e *invariante*. Sin embargo, la Real Academia de Ciencias Exactas, Físicas y Naturales de España, en la 3.ª edición de su *"Vocabulario Científico y Técnico"* publicado en Madrid en 1996 por Espasa, sí acepta el término *invariancia*.
4. En el original **array**. Este término se suele traducir también en Latinoamérica por **arreglo** y en España por matriz, lista o tabla y está muy extendido su uso en inglés. (N. del RT.)
5. *N.del T.* En inglés en el original. «Can´t» se puede interpretar como *no se puede* y «jack-in-a-box» puede considerarse como *tentetieso*.

NOTAS AL CAPÍTULO 3

1. El término *smooth* significa ajuste, aplanamiento, nivelación, aproximación, uniformidad, suavizamiento. Las funciones de este tipo se suelen conocer por *funciones suaves ,uniformes,* de *ajuste* o de *nivelación* (en escasas ocasiones "armónicas").

NOTAS AL CAPÍTULO 4

1. En inglés **while** (N. del T.).
2. El término inglés **loop** se suele traducir también por **lazo** o **ciclo** (N. del T.)

3. En inglés **for** (N. del T.)
4. En inglés **write** (N. del T.)

NOTAS AL CAPÍTULO 5

1. El término *array* es probablemente uno de los términos de difícil traducción al español, *vector* y *tabla* son términos aceptados. En numerosas obras en español se conserva el término en inglés. En casi toda Latinoamérica, se traduce por el término *arreglo*. Hemos optado por utilizar el término *matriz*, siendo conscientes de que no es la traducción más acertada.
2. Se suelen llamar estructuras LIFO, del inglés *last in, first out*. (N. del RT.)
3. Se suelen denominar estructuras FIFO, *first in, first out*. (N. del T.)
4. Los autores presuponen un tipo de datos incorporado o definido por el usuario denominado *complejo* y que representa a los números complejos. (N. del T.)
5. En Latinoamérica, el término *pointer* se suele traducir por *apuntador*. (N. del RT.)
6. El término inglés es *new*. (N. del RT.)

NOTAS AL CAPÍTULO 6

1. *Greedy Algorithms*, en el original. (N. del RT.)
2. El término inglés *Speanning* se suele traducir también envergadura. (N. del RT.)
3. Siguiendo la humorada del autor, podríamos quizá hablar de *reales*. (N. del T.)

NOTAS AL CAPÍTULO 10

1. El término probabilista se suele traducir también por probabilístico, aunque este término no es aceptado por el DRAE (Diccionario de la Real Academia de la Lengua Española). (N. del RT.)
2. El término primalidad no es aceptado por el DRAE. Se mantiene en la obra por su uso en la jerga matemática. (N. del RT.)
3. El término smooth se traducen por uniforme; también puede utilizarse el término aproximación, ajuste, suave, etc. (N. del RT.)

NOTAS AL CAPÍTULO 11

1. "CREW" son las siglas de concurrent-read, exclusive-write. Preferimos mantener el término original por cuanto está razonablemente extendido. (N. del T.)

NOTAS AL CAPÍTULO 12

1. "Nick's Class" en el original. (N. del T.)

NOTAS AL CAPÍTULO 13

1. El término *spanning* se suele traducir también por *envergadura* o *recubrimiento*. (N. del RT.)

Índice analítico

÷, 15

A

Aanderaa, S.O., 525
Abramson, B., 364
Aceleración óptima, 428
Ackermann, función de, 203, 208-9, 317
Ackermann, W., 209
Acondicionamiento previo, 327
Adel'son-Vel'skiĭ, G.M., 209
Adleman, L.M., 280, 289, 417, 523
Adyacencia máxima, 542
Adyacencia, matriz de, 174
Aflorar, 186
Aguja de Buffon, 371-5
Ahmes, 62
Aho, A.V., 62, 209, 364, 522-3
Akl, S.G., 62, 459
Al menos cuadrático, 480
al-Khowârizmî, 2
Alfa-beta, corte, 357
Alford, W.R., 417
Algorítmica, definición de, 3
Algoritmo, 2-7
  aproximado, 2, 525
  codificador, 280
  de Las Vegas, 393-408
  de Monte Carlo, 379-93
  de tiempo polinómico, 493
  eficiencia de un, 67
  eficiente, 491
  heurístico, 3, 525
  no determinístico, 508
  paralelo óptimo, 428
  probabilístico, 2
  sesgado, 390
  suave, 480
  voraz, características de un, 213
    características de un, 366
    numérico, 371-9
Algoritmo codificador, 280
Algoritmo de Dijkstra, 223-7, 245, 303
Algoritmo de Euclides, 2, 82, 89, 122
Algoritmo de Floyd, 303, 489
Algoritmo de Kruskal, 217-20
Algoritmo de Las Vegas, 393-6
Algoritmo de Monte Carlo, 379-93
Algoritmo de Prim, 220-3, 245
Algoritmo de Strassen, 490
Algoritmo de tiempo polinómico, 492
Algoritmo de Warshall, 314, 490
Algoritmo eficiente, 491
Algoritmo no determinístico, 508
Algoritmo paralelo óptimo, 429
Algoritmo suave, 480
Algoritmos numéricos probabilísticos, 371-9
Alicia, 280
Allison, L., 364
Altura media de un árbol, 464
Altura, 180
Amortizado, análisis, 127-32
  de algoritmos divide y vencerás, 251-2
  de estructuras de control, 111-8
  en caso peor, 70-1
  por término medio, 70-1, 126
    quicksort, 260
Amplificación, 390
Análisis de caso medio, 70, 126
Análisis en caso peor, 70
Anglicano, devocionario, 2, 90
Antecesor, 176
Antepasado, 176, 326
Aproximación absoluta, problema de, 538
Aproximación, esquema de, 545
Aproximación, problema de la,
  absoluta, 536
  relativa, 537
Aproximado, algoritmo, 525
Árbol, 175
  2-3, 180
  altura, 180
    media,
  AVL, 180
  binario, 178
    esencialmente completo, 184
  binomial, 193
  biselado, 187

búsqueda, 179
con raíz, 175
de Fibonacci, 198, 207
k-ario, 178
libre, 175
rojo-negro, 180
Árbol 2-3, 180
Árbol **AVL**, 180
Árbol binario, 178, 464
cálculo con un, 422
Árbol binomial, 193
Árbol biselado, 181
Árbol de búsqueda, 179, 317
óptimo, 316-7
Árbol de decisión, 462-3
válido, 466
Árbol de envergadura mínima, 215
Árbol de Fibonacci, 198, 207
Árbol de recubrimiento mínimo, 215-23, 531
Árbol rojo-negro, 180
Árboles de Steiner, 245
Argumentos antagónicos, 472-6
Argumentos teóricos de información, 462-72
Arista, 174
Aritmética modular, 279
Arlazarov, V.L., 522
Arora, S., 554
Articulación, puntos de, 332-4
Ascendencia, 328
Ascendente, 291, 308
Asignación, problema de, 349-52
Asintótica, notación, 70, 91-106
con varios parámetros, 104
condicional, 101
de orden exacto, 100
Omega, 98
alternativa, 107
operaciones sobre, 105
orden de, 91-2
Theta, 100
Asociativa, tabla, 181
Axioma del entero mínimo, 22

B

Baase, S., 62
Babai, L., 416, 523
Bach, E., 418
Bachmann, P.G.H., 110
Baeza-Yates, R., 62, 209
Ball, W.W.R., 364
Barajado perfecto, 446
Barómetro, 118
Batcher, K.E., 448, 459
Baumert, L., 364
Beauchemin, P., 417
Bellare, M., 523, 554
Bellman, R.E., 317
Bellmore, M., 364
Benioff, P., 523
Benito, 281
Bennett, C.H., 289, 523
Bentley, J.L., 89, 166, 289, 418
Berge, C., 209, 363
Berlekamp, E.R., 363
Berliner, H.J., 364
Bernstein, E., 523
Berthiaume, A., 523
Bezzel, 364
Bicoherente, 332
Biconexo, 332
Bishop, E., 62
Bitton, D., 459
Blum, L., 416
Blum, M., 289, 416-7, 523
Borodin, A.B., 62, 289, 523
Boruvka, O., 245
Brassard, G., 62, 89, 110, 288-9, 364, 416-8, 522-3
Bratley, P., 62, 89, 289, 364, 417-8, 522
Brent, R.P., 430, 459
Bressoud, D.M., 417-8
Brigham, E.O., 62
Brown, C.A., 166, 318
Brown, M.R., 209
Bucles **mientras**, 116
Bucles **para**, 112
Bucles **repetir**, 116
Bunch, J., 89, 522
Buneman, P., 166
Búsqueda binaria, 116, 255-7
Búsqueda en profundidad, 329-32
Búsqueda secuencial, 255

C

Cajas, problema de llenado de, 534-6, 546
Calinger, R., 63
Camarilla, 520
Cambio de variable, 148
Camino de Euler, 361
Camino Hamiltoniano, 361, 530
Camino simple, 298, 303
Caminos mínimos, 223-7, 301-4, 431, 487-90
Campeonato mundial, 293
Campo, 171
Carasso, C., 417
Cardinalidad, 9
Carente de istmos,
Carlsson, S., 209, 522
Carter, J.L., 417
Celis, P., 417
Centinela, 258
Certificado, 494
Chandra, A.K., 459, 523
Chang, L., 245, 317
Chen, I-N., 459
Cheriton, D., 245
Chin, F.Y., 459
Christofides, N., 62, 363, 553
Chvátal, V., 459
Ciclo Hamiltoniano, 530, 540
Cierre transitivo, 489
Circuito conmutador, 288
Clase de funciones universal2, 403
Clases de complejidad paralela, 514
Clases de complejidad probabilística, 513
Clases de complejidad, 511-5
paralela, 514
probabilística, 513
Cláusula, 502
Clave, 280
Co-NP, definición, 513
Cobham, A., 523
Cociente, 15
Codificación dispersiva, véase Tablas de Dispersión

Códigos de Huffman, 245
Coeficiente binomial, 47, 161, 292
Cohen, H., 417
Cola, 168, 205
Cole, R., 449, 459
Colisión, 182
Coloreado, 505
Combinación, 44
Comparador, 443
Complejidad, 100
de la ordenación, 466
Componentes conexos, 432-8
Compresión de rutas, 202
Comprobación de la conectividad de un grafo, 474
Comprobación de primalidad, 383-7
Computación admitida, 509
Computación distribuida, 455
Computadora cuántica, 281, 515
Conectado a dos bordes, 332
Constante de Euler, 44
Constante oculta, 70
Contador binario, 128, 130-1
Conteo probabilista, 377-9
Conteo probabilístico, 377-9
Conway, J.H., 363
Cook, S.A., 502, 522
Cooley, J.M., 85, 90
Coppersmith, D., 274, 289
Cormen, T.H., 62
Cortar, 440
Couvreur, C., 417
Crépeau, C., 417
Criba cuadrática, 408
Criba de campos numéricos, 407
Criptografía de clave pública, 280
Cuantificador, 11
Curtiss, J.H., 417
Curva elíptica, 408

D

Damgård, I.B., 417
Danielson, G.C., 85, 90
Dar la vuelta, 211, 295-9
de Bruijn, N.G., 110
De Moivre, 32, 75, 83, 137
Definición de PSPACE, 462
Definición de reducción, 477
lineal, 476-90
polinómica, 496-500
Demostración por contradicción, 15-18
indirecta, 15
no constructiva, 18
por inducción matemática, 18-35
Denenberg, L., 62, 209
Denning, D.E.R., 289
Deo, N., 62, 363, 459
Descendiente, 176
Desviación estándar, 55
Determinantes, 125
Deutsch, D., 523
Devocionario Anglicano, 2, 90
Devroye, L., 416
Dewdney, A.K., 166
DeWitt, D.J., 459
Deyong, L., 364
Dhall, SK., 62, 459
Diffie, 280, 289
Dijkstra, E.W., 245
Dinic, E.A., 525
Distribución *t* de Student, 375
Divide y vencerás, 446, 448
Dixon, J. D., 417
Dreyfus, S.E., 317
Dromey, R.G., 62
Duncan, R., 459
Duplicación de punteros, 424-7, 435

E

Ecuación característica, 135
Edmonds, J., 245, 523
Eficiencia de algoritmos paralelos, 428-31
Eficiente en términos de trabajo, 428
Einstein, 20
Ekert, A.K., 289
Eliminación de Gauss-Jordan, 78, 406
Elkies, N.D., 16, 62
Enorden, 326
Erdős, P., 417
Esencialmente completo, 184
Espacio de muestra, 48
Espacio de prueba, 494
Esperanza, 53
Estructura de conjunto disjunto, 198-204, 219, 239, 426, 432
Euclides, 16, 21
Euler, 19, 62-3, 383
Evaluación de expresiones en paralelo, 438-43
Evaluación de expresiones paralela, 438-43
Even, S., 62, 363
Eventualmente no decreciente, 103
Eves, H., 62, 63
Exponenciación, 274-8

F

Factor de carga, 183
Factorización de enteros muy grandes, 404-8, 456
Faradžev, I.A, 522
Fecha de Pascua, 56, 86, 89
Fermat, 19, 383
Feynman, R., 523
Fibonacci, 34, 63
Filtrar, 188
Fischer, MJ., 318, 522
Flajolet, Ph., 417
Flotar, 186
Floyd, R.W., 289, 317, 417
Forma normal conjuntiva, 502
Fórmula de Stirling, 15
Fox, B.L., 416
Fredman, M.L., 209, 245, 317, 522, 523
Freivalds, R., 417
Frye, 19
Fuertemente cuadrático, 481
Función, 11

Función de dispersión, 182
Función de Evaluación, 355
Función de memoria, 311, 322
Función suave, 103
Funciones potenciales, 129
Funciones, véase programas
Furman, M. E., 522
Fusión con ayuda, 451
Fusión de redes, 446-8

G

Galil, Z., 209
Gardner, M., 166, 289, 523
Garey, M.R., 245, 523, 553
Gauss, 85, 90
Generación de permutaciones, 346
Generador de números aleatorios, 369
Generador pseudoaleatorio, 370
Gibbons, A., 459
Gilbert, E.N., 318
Gill, J., 523
Godbole, S., 317
Goemans, M.X., 554
Goldner, A., 364
Goldreich, O., 523
Goldwasser, S., 418, 523
Golomb, S., 364
Gondran, M., 62, 363
Gonnet, G.H., 62, 209, 522
Good, I.J., 364
Goutier, C., 417
Grafo, 173
  acíclico, 336
  bipartito, 528
  conexo, 173
  fuertemente conexo, 173, 360
  planar, 360, 550
Grafo aciclico, 336
Grafo bipartito, 528
Grafo implicito, 342
Grafo planar, 360
Grafos fuertemente conexos, 360
Grafos y juegos, 319-26
Graham, R.L., 245
Granville, A., 417
Greene, D.H., 166
Gries, D., 89, 289
Guerra decimal, 274
Guibas, L.J., 209
Guy, R.K., 363

H

Haken, D., 166
Hall, A., 416
Halley, 20
Hammersley, J. M., 417
Handscomb, D.C., 417
Hanoi, Torres de, 124, 143
Hardy, G.H., 417
Harel, D., 62
heapsort, 79
Heath, T.L., 89
Held, M., 318
Hell, P., 245
Hellman, M.E., 280, 289
Hermano, 176
Heurístico, 3, 525
Hijo, 176
Hillier, F.S., 364
Hinrichs, K., 62
Hirschberg, D.S., 459
Hoare, C.A.R., 79, 260
Hoja, 176
Hopcroft, J.E., 62, 89, 209, 274, 289, 317, 364, 522, 523
Horowitz, E., 62, 209, 245, 553
Hsiao, DK., 459
Hu, T.C., 317, 364
Huang, M.- D.A., 417

I

Ibarra, O.H., 554
Inducción constructiva, 31-5, 264, 271
Inducción constructiva, 31-5, 264, 272, 310, 327
  generalizada, 28
  matemática, 18-35
Inducción matemática generalizada, 28
Iniciación virtual, 120, 169, 312, 323
Instancias, 66
  tamaño de las, 67
Integración cuasi Monte Carlo, 376
Integración numérica, 375-7
Integral múltiple, 376
Intervalo, 10
Intervalo de confianza, 367, 374
Introducción a la criptografía, 279-81
  clave pública, 280
  sistema RSA, 280
Irving, R.W., 364
Italiano, G.F.,

J

Ja'Ja'J., 62
Janko, W., 418
Jarník, V., 245
Jensen, K., 62
Johnson, D.B., 209, 245
Johnson, D.S., 245, 523, 554
Jozsa, R., 523
Juego de Grundy, 363
Juego de Marienbad, 319
Juego determinístico, 324
Juego simétrico, 324
Juegos determinísticos, 324
  simétricos, 324
Juegos y grafos, 319-26
Jugada, 520

K

K- suave, 405
Kahn, D., 289
Kaliski, B.S, 417
Karatsuba, A., 62, 89, 288
Karp, R., 209, 318, 522
Kasimi, T., 318
Kasparov, G., 364
Kerr, L.R., 274, 289

Kilian, J., 417
Kim, C.E., 554
Kim, S.H., 417
Kingston, J.H., 62, 209
Kirkerud, B., 89
Kleene, S.C., 317
Kline, M., 62
Knuth, D.E., 62, 89, 110, 166, 209, 289, 318, 364, 416, 417, 459
Koblitz, N., 289, 417, 418
König, H., 85, 90
Korsh, J., 245, 317
Kozen, D.C., 62, 209
Kraitchik, M., 417
Kranakis, E., 289, 417
Kronrod, M.A., 289, 522
Kruskal, J.B. Jr, 245
Kučera, L., 459

L

L'Écuyer, P., 416
Lakshmivarahan, S., 62, 459
Lam, J., 459
Lanczos, C., 85, 90
Landis, E.M., 209
Landrock, P., 417
Laporte, G., 553
Larson, P.-A., 417
Laurière, J.L., 317
Lawler, F.L, 62, 363, 364, 554
Lecarme, O., 62
Leclerc, G-L., 371, 416
Leighton, F.T., 62, 459
Leiserson, C.E., 62
Lenstra, A.K., 417, 418, 456, 459
Lenstra, H.W. Jr, 417, 418
Levin, G., 89, 289
Levin, L., 502, 523
Levy, L., 166
Lewis, H.R., 62, 209
Lewis, P.A., 89
lg, definición, 14
Lieberman, G.J., 364
Límites, 35
Linealmente equivalente, 480
Linealmente reducible, 480
Lista, 171, 424
Literal, 502
Llamadas recursivas, 114
Lloyd, S., 523
Logaritmo, 14
  por iteración, 208
LOGSPACE, definición, 512
Lucas, É., 166
Lueker, G.S., 166
Lund, C., 554

M

Manasse, M.S., 418, 456, 459
Manber, U., 62
Manders, K., 523
Máquina de von Neumann, 419
Máquina paralela de acceso aleatorio, 419-22
Marsh, D., 318
Martin, G.N., 417
Matriz, 167
Maurer, U., 417
Máximo común divisor, 82
Máximo de una matriz, 473
McCarty, C.P., 364
McDiarmid, C.J.H., 209
McGaughey, M., 364
Media, 53
Media jugada, 320
Mediana, aproximación, 269
  búsqueda, 266-72
  complejidad de búsqueda de la, 475
  definición, 267
Medida de la probabilidad, 49
Melhorn, K., 62, 363
Memoria de sólo escritura, 410
Mendeleev, 20
Menon, J., 459
mergesort, 79, 289
Merkle, R.C., 280, 289
Metodología descendente, 291, 309
Metropolis, I.N., 416
Meyer, A.R., 522
Micali, S., 416, 523
Michie, D., 318
Miller, G.L, 386, 417, 418
Minoux, M., 62, 363
Misère, 324
Modelo **CREW**, 421
Modelo de escritura exclusiva y lectura concurrente, 421
Modelo de instrucción única y múltiples flujos de datos, 421
Modelo de múltiples instrucciones y múltiples chorros de datos, 455
Modelo **F**, 420
Modelo **MIMD**, 455
Monet, S., 89, 288
Monier, L., 417
Montículo, 184-98
  binomial, 193-8
    retardado, 197
  de dos extremos, 198
  de Fibonacci, 198, 243
  invertido, 193
  k-ario, 207, 243
Montículo, 198
Montículo binomial, 193-8
  retardado, 198
Montículo de dos extremos, 198
Montículo de Fibonacci, 198, 243
Montículo invertido, 192
Montículo *k*-ario, 207, 243
Montreal, 298
Moore, E.F., 318
Moran, S., 523
Moret, B.ME., 62
Morris, R., 417
Motwani, R., 554
Multiplicación, 4-7
  à la russe, 4, 30, 62
  árabe, 56, 62
  clásica, 4
  de grandes enteros, 80, 247-51
  de matrices, 272
  divide y vencerás, 247
Multiplicación de matrices, 272

encadenada, 304-9
verificación, 380-3
Multiplicación encadenada de matrices, 304-9
Munro, J.I., 62, 209, 289, 417, 522, 523

N

Naik, A.V., 523
Nebut, J.-L., 62
Nelson, G., 209
Nemhauser, G., 317, 364
Newton, 47
Nievergelt, J., 62, 363
Nilsson, N., 62, 363
Nim, 319, 363
Nivel, 180
No articulado, 332
Nodo, 172
altura, 180
interno, 176
nivel, 180
profundidad, 180
Nodo interno, 176
Notación de punto, 171
Notación de teoría de conjuntos, 9
Notación del cálculo proposicional, 8
NP, definición, 494
NP-completo, 491-511
NP-completo, definición, 500
NP-difícil, definición, 507
Número cromático, 505, 540
Número de Catalán, 306, 314
Números naturales, 10

O

Ofman, Y., 62, 89, 288
Ogiwara, M., 523
Omar Khayyam, 47
Operación elemental, 73
Operación elemental, 73
Opolka, H., 63
Oppen, D.C., 209
Oráculo, 496
Orden exacto, 100
Orden, 70
Ordenación, 257
basada en comparaciones, 466
complejidad, 466-70
heapsort, 79
paralela, 449-53
pigeonhole, 79
por fusión, 79, 258-60
por inserción, 71, 121
por selección, 71, 120
probabilística, 400
quicksort, 79, 260-6
topológica, 336
Ordenación paralela, 449-53
Ordenación por casillas, 80
Ordenación por fusión paralela de Cole, 449
Ordenación por inserción, 71, 120
Ordenación por selección, 71, 120
Ordenación topológica, 336

P

P, definición, 493
P-correcto, 380
P-ram, 419-22
P-ram **CRCW**, 454
P-ram **EREW**, 454
Pageau, M., 417
Pan, V., 274
Papadimitriou, C.H., 63, 363, 522, 523
Par ordenado, 10
Parte entera, 14
Peralta, R.C., 418
Permutación, 45
Peso de monedas, 470
Pila, 168, 172
Pippenger, N., 522, 523
Pisano, Leonardo, véase Fibonacci
Pitágoras, 17
Pivote, 261
Planificación,
a fecha fija, 233
Polinomio característico, 136
Pólya, G., 21, 62
Pollard, J.M., 288, 418
Pomerance, C., 417, 523
Poner en cola, 168
Poner, 168
Posición terminal, 324
Postorden, 326
Pratt, V.R., 289, 523
Predicado, 11
Preorden, 326
Prim, R.C., 245
Principio cero-uno, 445
Principio de invariancia, 69
Principio de Minimax, 354-7
Principio de optimalidad, 298
Probabilidad condicional, 50
Problema de aproximación relativa, 538
Problema de coloreado de grafos, 491, 505, 526-8
Problema de Collatz, 364
Problema de corte máximo, 542
Problema de la mochila, 227-31, 299-301, 343-5, 353, 491, 532-4, 547-50
Problema de la partición, 521
Problema de las ocho reinas, 345-8, 396-400
Problema de satisfacibilidad, 502
Problema de selección, 266
Problema del ciclo Hamiltoniano, 491, 498, 510
Problema del embaldosado, 23
Problema del racimo mínimo, 542
Problema del viajante, 456, 491, 500, 528, 540
Euclídeo, 530
métrico, 529-32, 538
Problema suave, 480
Problemas, 66
suaves, 480
Problemas de aproximación NP-difícil, 536-41
Problemas de decisión, 492
Problemas NP-completos, 500-7
Procedimientos, véase Programas

Producto Cartesiano, 10
Profundidad, 180
Profundidad de una red, 444
Programas:
añadir-nodo, 189
aplanar, 426
Blancas, 356
biniter, 257
binrec, 256
buscar, 179
buscar 1, 199
buscar 2, 199
buscar 3, 203
buscar-max, 188, 454
buscar-raíz, 454
busquedabin, 256
Búsqueda_binaria, 116
C, 161, 292
caminospar, 431
card, 410
casilla, 80
compcons, 437
contar, 128, 378
crear-montículo, 190
crear-montículo-lento, 189
cuadrado, 21, 478
Decidir X, 499
desperdiciar, 161
devolver cambio, 212
dibujar, 414
Dijkstra, 224
distpar, 424
DV, 161, 252
estúpida, 390
Euclid, 83, 122
EvitarEuclides, 17
evalp, 441
expoDV, 276
expoiter, 278
expomod, 279
exposec, 274
Fermat, 384
Fibiter, 84, 113
Fibonaci, 33, 75
Fibrec, 83, 117
flotar, 188
Floyd, 303
fm, 309
fm-mem, 312
Freivalds, 381
FreivaldsEpsilon, 382
fusionar, 258
fusionar 1, 199
fusionar 2, 199
fusionar 3, 201
ganadin, 322
ganarec, 322
Ham, 497
HamD, 497
HamND, 510
Hanoi, 124
hundir, 188
índicemax, 473
iniciar, 378
insertar, 71, 121
intDET, 376
intMC, 375
Kruskal, 220
MAX-CORTAR-*aprox*, 543
MC3, 390
mcd, 82
MillRab, 386
mochila, 228
mochila-aprox, 533
mochila-voraz, 533
mochilava, 344
modificar-montículo, 187
monedas, 297
mult, 478
Negras, 356
nim, 323
Nuevoprimo, 16
ordenarporfusión, 258
ordenación por montículo, 191
operpar, 426
ordenacioninsitu3, 467
P, 293
perm, 346
pivote, 261
pivotebis, 266
Prim, 221, 222
primoaleatorio, 388
pruebaB, 285
pruebadiv, 404
pseudomediana, 269
pulsar, 378
quicksort, 261
quicksortLV, 402
ra, 339
recorridop, 330
reinas, 347
*reinas*1, 345
*reinas*2, 346
reinasLV, 397
RepetirFreivalds, 382
RepetirLV, 395
RepetirMC, 391
RepetirMillRab, 387
rp, 330
*rp*2, 338
rusa, 8
selección, 268
seleccionar, 71, 120
selecciónLV, 401
secuencia, 237
*secuencia*2, 240
secuencial, 255
serie, 295
Sum, 75
sumapar, 423
tiramoneda, 369
vueltaatrás, 348
Wilson, 74
XND, 511
Programas, notación, 8
notación paralela, 421
Propiedad, 11
Propiedad del montículo, 185
Propiedad Métrica, 529
Pseudoprimo estricto, 386
Pseudoprimos, 386
Puntero, 171
Purdom, P.W. Jr, 166, 318

Q

quicksort, 79, 260-6, 289
Quinn, M.J., 459
Quisquater, J.-J., 417
Quitar, 168
Quitar de cola, 168

R

Rabin, M.O., 386, 417, 418
Racimo, 542
Rackoff, C., 523
Raíz, 176
Ramificación y poda, 348-54
consideraciones generales, 354
Rawlins, G.J.E., 166
Razón áurea, 32, 76
Reconstrucción de tablas de dispersión, 182
Recorrido en anchura, 337-42
Recuento, 288
Recurrencias, asintóticas, 157
cambio de variable, 148

ecuación característica, 135
lineal, homogéneo, 135
inhomogéneo, 140
polinomio característico, 137
resolución, 132-160
transformaciones de intervalo, 156
Red de ordenación por inserción, 444
Red de ordenación por selección, 444
Redes de ordenación, 443-5
Redes par-impar, 448
Redes, profundidad, 444
tamaño, 444
Reducción lineal, 476-90
Reducción polinómica, 496-500
Reducciones entre problemas matriciales, 482-7
entre problemas de camino mínimo, 487-90
Reducible a decisión, 498
Reed, B.A., 209
Registro, 170
Regla de dualidad, 98
Regla de l'Hôpital, 38
Regla de suavidad, 104
Regla del Límite, 96, 100
Regla del máximo, 94-8
Regla del umbral, 93, 100
Reif, J.H., 459
Reingold, E.M., 62, 363
Relación, 11
Retroceso (*backtracking*), 342-8
caso general, 348
Rice, J.A., 416
Rivest, R.L., 62, 280, 289, 417
Robertson, G.G., 459
Robin Hood, efecto, 394
Robson, J.M., 364
Rosen, K.H., 62
Rosenthal, A., 364
Rumely, R.S., 418
Runge, C., 85, 89
Rytter, W., 459

S

Sahni, S., 62, 209, 245, 554
Sarwate, D.V., 459
Satisfacible, 502
Savitch, W.J., 523
Saxe, J.B., 166
Schaffer, R, 209
Scharlau, W, 63
Schneier, B., 289
Schönhage, A., 89, 288
Schrage, L.E., 417
Schwartz, E.S., 245
Schwartz, J., 418
Sedgewick, R., 62, 209
Selección de un umbral:
empíricamente, 254
enfoque híbrido, 254
Selección probabilística, 400
Selman, A.L., 523
Semilla, 370
Series aritméticas, 39
armónicas, 43
geométricas, 40
Sesgado, algoritmo, 390
Shallit, J., 416, 418
Shamir, A., 89, 280, 289
Shapiro, H.D., 62
Sherman, A.T., 417
Shing, M.T., 317
Shor, P.W., 289, 523
Shub, M., 416
Simmons, G.J., 289
Simon, D.R., 523
Simplificación, 252
Simulación, 370
Sistema criptográfico RSA, 280
Sistema de prueba, 494
Sleator, D.D., 209
Sloane, N.J.A., 318
Sobol', I.M., 416
Solomon, H., 416
Solovay, R., 417
Stanat, D.F., 417
Standish, T.A., 209
Steele, J.M., 417
Steiglitz, K., 62, 363
Stinson, D.R., 62, 289
Stockmeyer, LJ., 523
Stone, H.S., 209
Strassen, V., 78, 89, 273, 288, 289, 417
Subalgoritmo básico, 251
Sucesión de Fibonacci, 32, 83, 137
Suceso, 48
Sudan, M., 554
Suma condicional, 13
Sumador, 288
Sumador completo, 288
Sumidero, 361
Szegedy, M., 554

T

Tabla asociativa, 181
Tabla de símbolos, 182
Tablas de dispersión, 182
universales, 402
Tablas dispersivas universales, 402
Tamaño de una red, 444
Tarjan, R.F., 62, 166, 204, 209, 245, 289, 363, 364
Techo, 15
Teorema Central del Límite, 56, 373, 392
Teorema de Brent, 430
Teorema de los Números Primos, 51, 411
Teorema de Wilson, 74, 88
Teorema fundamental de la aritmética, 28
Teorema pequeño de Fermat, 383
Testigo falso, 385
Testigo falso estricto, 386
Testigo falso estricto, 386
Testigo falso, 384
estricto, 386
Texto cifrado, 280
Tiempo esperado, 368
Tiempo polilogarítmico, 428
Tiempo promedio, 368
Tipo:
adjgraph, 174
binary-node, 179
K-ary-node, 178
lisgraph, 175
tablelist, 181

treenode 1, 177
treenode 2, 177
Torneo, 287
Torres de Hanoi, 124, 143
Trabajo, 428-31
Transformaciones de intervalo, 156
Transformada rápida de Fourier, 85
Tres en raya tridimensional, 364
Triángulo de Pascal, 292
Truco de contabilidad, 131,197
Tucker, L.W., 459
Tukey, J.W., 85, 89
Turing, A.M., 523
Turk, J.W.M., 289

U

Ulam, S., 416
Ullman, J.D., 62, 209, 317, 364, 522, 523
Umbral, 92-3
  para divide y vencerás, 552
Urbanek, F.J., 89, 289

V

Van Leeuwen, J., 62
Variable aleatoria, 54
Varianza, 55
Vazirani, U.V., 417, 523
Veinte preguntas, 462
Ventaja, 390
Veredicto, 463
Verificación de la multiplicación de matrices, 380-3
Verma, R.M., 166
Vickery, C.W., 417
Von Neumann, J., 416, 419
Vuillemin, J., 209

W

Wagner, R.A., 318
Warshall, S., 317
Warusfel, A., 62
Wegman, M.N., 417
Welch, PD., 89
Wigderson, A., 523
Willard, D.E., 522
Williams, H., 417
Williamson, D.P., 554
Willians, J.W.J., 79, 190, 209
Winograd, S., 274, 289, 523
Winter, P., 245
Wirth, N., 62
Wood, D., 209
Wood, D.W., 364
Wright, E.M., 417
Wright, J.W., 245, 317

Y

Yao, A.C., 416
Yao, F.F., 318
Yee, C.N., 364
Yeo, A.C., 245
Younger, D.H., 318
Yung, M.M., 364

Z

Zippel, R.E., 417
Zuffellato, D., 89, 288