

TOMÁS H.

CORMEN

CARLOS E.

LEISERSON

RONALD

REMACHE

CLIFFORD STEIN

INTRODUCCIÓN A

# ALGORITMOS

TERCERA EDICION

---

# Introducción a los Algoritmos

Tercera edición



---

Thomas H. Cormen  
Charles E. Leiserson  
Ronald L. Rivest  
Clifford Stein

---

## Introducción a los Algoritmos

Tercera edición

La prensa del MIT  
Cambridge, Massachusetts Londres, Inglaterra

c 2009 Instituto de Tecnología de Massachusetts

Reservados todos los derechos. Ninguna parte de este libro puede reproducirse de ninguna forma ni por ningún medio electrónico o mecánico (incluidas las fotocopias, las grabaciones o el almacenamiento y la recuperación de información) sin el permiso por escrito del editor.

Para obtener información sobre descuentos especiales por cantidad, envíe un correo electrónico [a special sales@mitpress.mit.edu](mailto:a special sales@mitpress.mit.edu).

Este libro fue ambientado en Times Roman y Mathtime Pro 2 por los autores.

Impreso y encuadrado en los Estados Unidos de América.

Datos de catalogación en publicación de la Biblioteca del Congreso

Introducción a los algoritmos / Thomas H. Cormen. . [et al.]—3ra ed.

pag. cm.

Incluye referencias bibliográficas e índice.

ISBN 978-0-262-03384-8 (tapa dura: papel alcalino)—ISBN 978-0-262-53305-8 (pbk.: papel alcalino)

1. Programación informática. 2. Algoritmos informáticos. I. Cormen, Thomas H.

QA76.6.I5858 2009 005.1

—dc22

2009008593

10 9 8 7 6 5 4 3 2

# Contenido

Prefacio XIII

---

## Fundaciones

Introducción	3
1 El papel de los algoritmos en la computación	5
1.1 Algoritmos	5
1.2 Los algoritmos como tecnología	11
2 Primeros pasos	16
2.1 Clasificación por inserción	
16.2.2 Análisis de algoritmos	23
2.3 Diseño de algoritmos	29
3 Crecimiento de funciones	43
3.1 Notación asintótica	43
3.2 Notaciones estándar y funciones comunes	53
4 Divide y vencerás	65
4.1 El problema del subarreglo máximo	68
4.2 El algoritmo de Strassen para la multiplicación de matrices	
75	
4.3 El método de sustitución para resolver recurrencias	
83	
4.4 El método del árbol de recurrencias para resolver recurrencias	
88	
4.5 El método maestro para resolver recurrencias	
93	
4.6 Demostración del teorema del maestro	97
5 Análisis probabilístico y algoritmos aleatorios	114
5.1 El problema de la contratación	
114	
5.2 Indicador de variables aleatorias	
118	
5.3 Algoritmos aleatorios	122
? 5.4 Análisis probabilístico y otros usos de variables aleatorias indicadoras	
130	

---

**II Estadísticas de Clasificación y Orden**

---

Introducción	147
6 Heapsort	151
6.1 Heaps	151
6.2 Mantenimiento de la propiedad heap	154
6.3 Creación de un montón	
6.4 El algoritmo heapsort	159
6.5 Colas de prioridad	162
7 Quicksort	170
7.1 Descripción de quicksort	170
7.2 Rendimiento de quicksort	174
7.3 Una versión aleatoria de quicksort	179
7.4 Análisis de clasificación rápida	180
7.5 Ordenar	
8 en tiempo lineal	191
8.1 Límites inferiores para ordenar	191
8.2 Ordenar por conteo	194
8.3 Ordenar por radix	197
8.4 Ordenar por cubo	200
9 Estadísticas de medianas y órdenes	213
9.1 Mínimo y máximo	214
9.2 Selección en tiempo lineal esperado	215
9.3 Selección en tiempo lineal del peor caso	220

---

**III Estructuras de datos**

---

Introducción	229
10 Estructuras de datos elementales	232
10.1 Pilas y colas	232
10.2 Listas enlazadas	236
10.3 Implementación de punteros y objetos	241
10.4 Representación de árboles con raíces	246
11 tablas hash	253
11.1 Tablas de direcciones directas	254
11.2 Tablas hash	256
11.3 Funciones hash	262
11.4 Direcccionamiento abierto	
11.5 Hashing perfecto	277

12 Árboles de búsqueda binarios 286
12.1 ¿Qué es un árbol de búsqueda binario? 286
12.2 Consultando un árbol de búsqueda binaria
289 12.3 Inserción y borrado 294 ? 12.4
Árboles de búsqueda binarios construidos aleatoriamente 299
13 árboles rojo-negros 308
13.1 Propiedades de los árboles rojo-negros 308
13.2 Rotaciones 312
13.3 Inserción 315
13.4 Eliminación 323
14 Aumento de estructuras de datos 339 14.1
Estadísticas de orden dinámico 339 14.2
Cómo aumentar una estructura de datos 345 14.3
Árboles de intervalo 348

---

## IV Técnicas Avanzadas de Diseño y Análisis

---

Introducción 357
15 Programación dinámica 359 15.1 Corte
de varillas 360 15.2
Multiplicación matriz-cadena 370 15.3 Elementos
de programación dinámica 378 15.4 Subsecuencia común
más larga 390 15.5 Árboles de búsqueda binarios
óptimos 397 16 Algoritmos codiciosos 414 16.1
Un problema de selección de
actividades 415 16.2 Elementos de la estrategia
codiciosa 4 23 16.3 Huffman códigos 428
? 16.4 Matroids y métodos codiciosos 437 ? 16.5 Un
problema de programación de tareas como matroid 443 17 Análisis
amortizado 451
17.1 Análisis agregado 452 17.2 El
método contable 456 17.3 El método
potencial 459 17.4 Tablas dinámicas 463

---

## V Estructuras de datos avanzadas

---

Introducción	481
18 B-Árboles	484
18.1 Definición de árboles B	488
18.2 Operaciones básicas en árboles B	491
18.3 Eliminación de una clave de un árbol B	499
19 Montones de Fibonacci	505
19.1 Estructura de los montones de Fibonacci	
507 19.2 Operaciones de montones combinables	510
19.3 Disminución de una clave y eliminación de un nodo	518
19.4 Limitación del grado máximo	523
20 de Emde Boas Árboles	531
20.1 Aproximaciones preliminares	532
20.2 Una estructura recursiva	536
20.3 El árbol de van Emde Boas	545
21 Estructuras de datos para conjuntos disjuntos	561
21.1 Operaciones de conjuntos disjuntos	561
21.2 Representación de listas enlazadas de conjuntos disjuntos	
564 21.3 Bosques de conjuntos disjuntos	568
? 21.4 Análisis de unión por rango con compresión de trayectoria	573

---

## Algoritmos de gráfico VI

---

Introducción	587
22 Algoritmos de grafos elementales	589
22.1 Representaciones de grafos	589
22.2 Búsqueda primero en amplitud	594
22.3 Búsqueda primero en profundidad	603
22.4 Clasificación topológica	612
22.5 Componentes fuertemente conectados	615
23 Árboles de expansión mínimos	624
23.1 Crecimiento de un árbol de expansión mínimo	625
23.2 Los algoritmos de Kruskal y Prim	631

24 Rutas más cortas de fuente única 643	
24.1 El algoritmo de Bellman-Ford 651	24.2
Caminos más cortos de fuente única en gráficos acíclicos dirigidos 655	24.3
Algoritmo de Dijkstra 658	24.4
Restricciones de diferencia y caminos más cortos 664	24.5
Pruebas de las propiedades de los caminos más cortos 671	
25 Rutas más cortas de todos los pares 684	
25.1 Caminos más cortos y multiplicación de matrices 686	25.2
El algoritmo de Floyd-Warshall 693	25.3 El algoritmo
de Johnson para gráficos dispersos 700	
26 Caudal máximo 708	
26.1 Redes de flujo 709	26.2 El
método Ford-Fulkerson 714	26.3 Coincidencia
bipartita máxima 732	
? 26.4 Algoritmos push-relabel 736	? 26.5 El
algoritmo de reetiquetado al frente 748	

---

## VII Temas Seleccionados

---

Introducción	769
27 Algoritmos de subprocessos múltiples 772	27.1
Los fundamentos de los subprocessos múltiples dinámicos 774	
27.2 Multiplicación de matrices de subprocessos múltiples	
792	27.3 Clasificación por combinación de
subprocessos múltiples 797	28
Operaciones con matrices 813	28.1 Resolución de sistemas
de ecuaciones lineales 813	28.2
Inversión de matrices 827	28.3 Matrices simétricas definidas positivas y mínimos cuadrados ap
832	aproximación
29 Programación lineal 843	29.1 Formas
estándar y de holgura 850	29.2 Formulación
de problemas como programas lineales 859	29.3 El algoritmo
símplex 864	29.4 Dualidad 879
La	29.5 La
solución factible básica	
inicial 886	

x

Contenido

30 Polinomios y la FFT 898

30.1 Representación de polinomios 900 30.2

La DFT y la FFT 906 30.3

Implementaciones eficientes de FFT 915

31 Algoritmos de teoría de números 926 31.1

Nociones elementales de teoría de números 927 31.2

Máximo común divisor 933

31.3 Aritmética modular 939

31.4 Resolver ecuaciones lineales modulares 946

31.5 El teorema chino del resto 950

31.6 Potencias de un elemento 954

31.7 El criptosistema de clave pública RSA 958 ? 31.8

Prueba de primalidad 965 ? 31.9

Factorización de enteros 975 32 Coincidencia

de cadenas 985 32.1 El algoritmo

ingenuo de coincidencia de cadenas 988 32.2 El

algoritmo de Rabin-Karp 990 32.3

Coincidencia de cadenas con autómatas finitos 995 ?

32.4 El algoritmo de Knuth-Morris-Pratt 1002 33 Geometría

computacional 1014

33.1 Propiedades de segmento de línea

1015 33.2 Determinación de si cualquier par de segmentos se interseca

1021 33.3 Hallazgo del casco convexo

1029 33.4 Hallazgo del par de puntos más cercano 1039

34 NP-Complejidad 1048

34.1 Tiempo polinomial 1053 34.2

Verificación de tiempo polinomial 1061 34.3 NP-

completitud y reducibilidad 1067 34.4 Pruebas de NP-

completitud 1078 34.5 Problemas NP-

completos 1086

35 Algoritmos de aproximación 1106

35.1 El problema de la cobertura de vértices

1108 35.2 El problema del viajante de comercio 1111

35.3 El problema de la cobertura de conjuntos

1117 35.4 Aleatorización y programación lineal 1123 35.5 El

problema de la suma de subconjuntos 1128

---

## VIII Apéndice: Antecedentes matemáticos

---

Introducción	1143
A Sumas	1145
A.1 Fórmulas y propiedades de sumatoria	1145
Sumatorias acotadas	1149
B Conjuntos, etc.	1158
B.1 Conjuntos	1158
B.2 Relaciones	1158
B.3 Funciones	1166
B.4 Gráficos	1168
Árboles	1173
C Conteo y probabilidad	1183
C.1 Conteo	1183
C.2 Probabilidad	1189
C.3 Variables aleatorias discretas	1196
C.4 Las distribuciones geométrica y binomial	1201
? C.5 Las colas de la distribución binomial	1208
D Matrices	1217
D.1 Matrices y operaciones con matrices	1217
Propiedades básicas de las matrices	1222

---

Bibliografía	1231
--------------	------

Índice	1251
--------	------



## Prefacio

Antes de que existieran las computadoras, existían los algoritmos. Pero ahora que hay computadoras, hay incluso más algoritmos, y los algoritmos se encuentran en el corazón de la computación.

Este libro ofrece una introducción completa al estudio moderno de los algoritmos informáticos. Presenta muchos algoritmos y los cubre con considerable profundidad, pero hace que su diseño y análisis sean accesibles para todos los niveles de lectores. Hemos tratado de mantener las explicaciones elementales sin sacrificar la profundidad de la cobertura o el rigor matemático.

Cada capítulo presenta un algoritmo, una técnica de diseño, un área de aplicación o un tema relacionado. Los algoritmos se describen en inglés y en un pseudocódigo diseñado para que cualquiera que haya hecho un poco de programación los pueda leer. El libro contiene 244 figuras, muchas con múltiples partes, que ilustran cómo funcionan los algoritmos. Dado que hacemos hincapié en la eficiencia como criterio de diseño, incluimos análisis cuidadosos de los tiempos de ejecución de todos nuestros algoritmos.

El texto está destinado principalmente para su uso en cursos de pregrado o posgrado en algoritmos o estructuras de datos. Debido a que analiza problemas de ingeniería en el diseño de algoritmos, así como aspectos matemáticos, es igualmente adecuado para el autoaprendizaje por parte de profesionales técnicos.

En esta, la tercera edición, una vez más hemos actualizado todo el libro. Los cambios cubren un amplio espectro, incluidos nuevos capítulos, pseudocódigo revisado y un estilo de escritura más activo.

### Al maestro

Hemos diseñado este libro para que sea versátil y completo. Debería encontrarlo útil para una variedad de cursos, desde un curso de pregrado en estructuras de datos hasta un curso de posgrado en algoritmos. Debido a que proporcionamos mucho más material del que puede caber en un curso típico de un trimestre, puede considerar este libro como un "buffet" o "smorgasbord" del que puede elegir el material que mejor se adapte al curso que desea realizar. enseñar.

Le resultará fácil organizar su curso en torno a los capítulos que necesita. Hemos creado capítulos relativamente autónomos, de modo que no tenga que preocuparse por una dependencia inesperada e innecesaria de un capítulo con respecto a otro. Cada capítulo presenta el material más fácil primero y el material más difícil después, con límites de sección que marcan puntos de parada naturales. En un curso de pregrado, puede usar solo las secciones anteriores de un capítulo; en un curso de posgrado, puede cubrir todo el capítulo.

Hemos incluido 957 ejercicios y 158 problemas. Cada sección termina con ejercicios y cada capítulo termina con problemas. Los ejercicios son generalmente preguntas cortas que evalúan el dominio básico del material. Algunos son simples ejercicios de pensamiento de autoevaluación, mientras que otros son más sustanciales y adecuados como tareas asignadas para el hogar. Los problemas son estudios de casos más elaborados que a menudo introducen material nuevo; a menudo consisten en varias preguntas que guían al estudiante a través de los pasos necesarios para llegar a una solución.

Partiendo de nuestra práctica en ediciones anteriores de este libro, hemos puesto a disposición del público soluciones para algunos, pero de ninguna manera todos, los problemas y ejercicios. Nuestro sitio web, <http://mitpress.mit.edu/algorithms/>, tiene enlaces a estas soluciones. Deberá revisar este sitio para asegurarse de que no contenga la solución a un ejercicio o problema que planea asignar. Esperamos que el conjunto de soluciones que publicamos crezca lentamente con el tiempo, por lo que deberá verificarlo cada vez que imparta el curso.

Hemos destacado (?) las secciones y ejercicios que son más adecuados para estudiantes de posgrado que para estudiantes de pregrado. Una sección con asterisco no es necesariamente más difícil que una sin asterisco, pero puede requerir conocimientos matemáticos más avanzados. Del mismo modo, los ejercicios destacados pueden requerir una formación avanzada o una creatividad superior a la media.

#### Para el estudiante

Esperamos que este libro de texto le proporcione una agradable introducción al campo de los algoritmos. Hemos intentado hacer que cada algoritmo sea accesible e interesante. Para ayudarlo cuando encuentre algoritmos desconocidos o difíciles, describimos cada uno de ellos paso a paso. También proporcionamos explicaciones detalladas de las matemáticas necesarias para comprender el análisis de los algoritmos. Si ya está familiarizado con un tema, encontrará los capítulos organizados para que pueda hojear las secciones introductorias y pasar rápidamente al material más avanzado.

Este es un libro grande y su clase probablemente cubrirá solo una parte de su material. Sin embargo, hemos tratado de hacer de este un libro que le sea útil ahora como libro de texto de curso y también más adelante en su carrera como referencia de escritorio matemático o manual de ingeniería.

¿Cuáles son los requisitos previos para leer este libro?

Debes tener algo de experiencia en programación. En particular, debe comprender los procedimientos recursivos y las estructuras de datos simples, como matrices y listas enlazadas.

Debe tener cierta facilidad con las demostraciones matemáticas, y especialmente con las demostraciones por inducción matemática. Algunas partes del libro se basan en algún conocimiento de cálculo elemental. Más allá de eso, las Partes I y VIII de este libro le enseñan todas las técnicas matemáticas que necesitará.

Hemos escuchado, alto y claro, el llamado a aportar soluciones a problemas y ejercicios. Nuestro sitio web, <http://mitpress.mit.edu/algorithms/>, contiene enlaces a soluciones para algunos de los problemas y ejercicios. Siéntase libre de comparar sus soluciones con las nuestras. Sin embargo, le pedimos que no nos envíe sus soluciones.

al profesional

La amplia gama de temas de este libro lo convierte en un excelente manual sobre algoritmos. Debido a que cada capítulo es relativamente independiente, puede concentrarse en los temas que más le interesen.

La mayoría de los algoritmos que discutimos tienen una gran utilidad práctica. Por lo tanto, abordamos las preocupaciones de implementación y otros problemas de ingeniería. A menudo proporcionamos alternativas prácticas a los pocos algoritmos que son principalmente de interés teórico.

Si desea implementar cualquiera de los algoritmos, debería encontrar que la traducción de nuestro pseudocódigo a su lenguaje de programación favorito es una tarea bastante sencilla. Hemos diseñado el pseudocódigo para presentar cada algoritmo de forma clara y sucinta. En consecuencia, no abordamos el manejo de errores y otros problemas de ingeniería de software que requieren suposiciones específicas sobre su entorno de programación. Intentamos presentar cada algoritmo de manera simple y directa sin permitir que las idiosincrasias de un lenguaje de programación en particular oscurezcan su esencia.

Entendemos que si está utilizando este libro fuera de un curso, es posible que no pueda comparar sus soluciones a problemas y ejercicios con las soluciones proporcionadas por un instructor. Nuestro sitio web, <http://mitpress.mit.edu/algorithms/>, contiene enlaces a soluciones para algunos de los problemas y ejercicios para que pueda verificar su trabajo. Por favor, no nos envíe sus soluciones.

A nuestros colegas

Hemos proporcionado una extensa bibliografía y sugerencias para la literatura actual. Cada capítulo termina con un conjunto de notas de capítulo que brindan detalles históricos y referencias. Las notas del capítulo no proporcionan una referencia completa a todo el campo.

de algoritmos, sin embargo. Aunque puede ser difícil de creer para un libro de este tamaño, las limitaciones de espacio nos impidieron incluir muchos algoritmos interesantes.

A pesar de las innumerables solicitudes de los estudiantes de soluciones a problemas y ejercicios, hemos optado por política de no proporcionar referencias para problemas y ejercicios, para eliminar la tentación de que los estudiantes busquen una solución en lugar de encontrarla por sí mismos.

#### Cambios para la tercera edición

¿Qué ha cambiado entre la segunda y la tercera edición de este libro? La magnitud de los cambios está a la par con los cambios entre la primera y la segunda edición. Como dijimos sobre los cambios de la segunda edición, dependiendo de cómo se mire, el libro cambió poco o bastante.

Un vistazo rápido a la tabla de contenido muestra que la mayoría de los capítulos y secciones de la segunda edición aparecen en la tercera edición. Eliminamos dos capítulos y una sección, pero hemos agregado tres nuevos capítulos y dos nuevas secciones además de estos nuevos capítulos.

Mantuvimos la organización híbrida de las dos primeras ediciones. En lugar de organizar los capítulos solo por dominios de problemas o de acuerdo solo con las técnicas, este libro tiene elementos de ambos. Contiene capítulos basados en técnicas sobre divide y vencerás, programación dinámica, algoritmos codiciosos, análisis amortizado, NP-Compleitud y algoritmos de aproximación. Pero también tiene partes enteras sobre clasificación, sobre estructuras de datos para conjuntos dinámicos y sobre algoritmos para problemas de gráficos. Encontramos que aunque necesita saber cómo aplicar técnicas para diseñar y analizar algoritmos, los problemas rara vez le indican qué técnicas son más adecuadas para resolverlos.

Aquí hay un resumen de los cambios más significativos para la tercera edición:

Agregamos nuevos capítulos sobre árboles de van Emde Boas y algoritmos multiproceso, y hemos desglosado el material sobre conceptos básicos de matrices en su propio capítulo de apéndice.

Revisamos el capítulo sobre recurrencias para cubrir más ampliamente la técnica de divide y vencerás, y sus dos primeras secciones aplican divide y vencerás para resolver dos problemas. La segunda sección de este capítulo presenta el algoritmo de Strassen para la multiplicación de matrices, que hemos trasladado del capítulo sobre operaciones con matrices.

Eliminamos dos capítulos que rara vez se enseñaban: montones binomiales y redes de clasificación. Una idea clave en el capítulo de clasificación de redes, el principio 0-1, aparece en esta edición dentro del problema 8-7 como el lema de clasificación 0-1 para comparar algoritmos de intercambio. El tratamiento de los montones de Fibonacci ya no depende de los montones binomiales como precursores.

Revisamos nuestro tratamiento de la programación dinámica y los algoritmos codiciosos. La programación dinámica ahora comienza con un problema más interesante, el corte de varillas, que el problema de programación de la línea de montaje de la segunda edición. Además, enfatizamos la memorización un poco más que en la segunda edición, e introducimos la noción del gráfico de subproblemas como una forma de comprender el tiempo de ejecución de un algoritmo de programación dinámica. En nuestro ejemplo inicial de algoritmos voraces, el problema de selección de actividad, llegamos al algoritmo voraz más directamente que en la segunda edición.

La forma en que eliminamos un nodo de los árboles de búsqueda binarios (que incluye árboles rojo-negro) ahora garantiza que el nodo cuya eliminación se solicita es el nodo que realmente se elimina. En las dos primeras ediciones, en ciertos casos, se eliminaba algún otro nodo, y su contenido se trasladaba al nodo pasado al procedimiento de eliminación. Con nuestra nueva forma de eliminar nodos, si otros componentes de un programa mantienen punteros a nodos en el árbol, no terminarán por error con punteros obsoletos a nodos que se han eliminado.

El material de las redes de flujo ahora basa los flujos completamente en los bordes. Este enfoque es más intuitivo que el flujo neto utilizado en las dos primeras ediciones.

Con el material sobre conceptos básicos de matrices y el algoritmo de Strassen trasladado a otros capítulos, el capítulo sobre operaciones con matrices es más pequeño que en la segunda edición.

Hemos modificado nuestro tratamiento del algoritmo de emparejamiento de cadenas de Knuth-Morris-Pratt.

Corregimos varios errores. La mayoría de estos errores se publicaron en nuestro sitio web de erratas de la segunda edición, pero algunos no.

En base a muchas solicitudes, cambiamos la sintaxis (por así decirlo) de nuestro pseudocódigo. Ahora usamos "D" para indicar la asignación y "==" para probar la igualdad, tal como lo hacen C, C++, Java y Python. Asimismo, hemos eliminado las palabras clave do y luego y hemos adoptado "//" como nuestro símbolo de comentario al final de la línea. Ahora también usamos la notación de puntos para indicar los atributos de los objetos. Nuestro pseudocódigo sigue siendo procedural, en lugar de estar orientado a objetos. En otras palabras, en lugar de ejecutar métodos en objetos, simplemente llamamos a procedimientos, pasando objetos como parámetros.

Agregamos 100 nuevos ejercicios y 28 nuevos problemas. También actualizamos muchas entradas de la bibliografía y añadimos varias nuevas.

Finalmente, repasamos todo el libro y reescribimos oraciones, párrafos y secciones para que la escritura fuera más clara y activa.

### Sitio web

Puede usar nuestro sitio web, <http://mitpress.mit.edu/algorithms/>, para obtener información adicional y comunicarse con nosotros. El sitio web tiene enlaces a una lista de errores conocidos, soluciones a ejercicios y problemas seleccionados y (por supuesto) una lista que explica los chistes cursis del profesor, así como otro contenido que podríamos agregar. El sitio web también le indica cómo informar errores o hacer sugerencias.

### Cómo producimos este libro

Al igual que la segunda edición, la tercera edición se produjo en LATEX 2". Usamos la fuente Times con tipografía matemática usando las fuentes MathTime Pro 2. Agradecemos a Michael Spivak de Publish or Perish, Inc., Lance Carnes de Personal TeX, Inc. y Tim Tregubov de Dartmouth College por el apoyo técnico. Como en las dos ediciones anteriores, compilamos el índice utilizando Windex, un programa C que escribimos, y la bibliografía se produjo con BIBTEX. Los archivos PDF de este libro se crearon en un MacBook con sistema operativo 10.5.

Dibujamos las ilustraciones para la tercera edición usando MacDraw Pro, con algunas de las expresiones matemáticas en las ilustraciones incluidas con el paquete psfrag para LATEX 2". Desafortunadamente, MacDraw Pro es un software heredado que no se comercializa desde hace más de una década. Felizmente , todavía tenemos un par de Macintosh que pueden ejecutar el entorno Classic en OS 10.4 y, por lo tanto, pueden ejecutar principalmente Mac Draw Pro. Incluso en el entorno Classic, creemos que MacDraw Pro es mucho más fácil de usar que cualquier otro software de dibujo. para los tipos de ilustraciones que acompañan al texto de informática, y produce resultados hermosos.<sup>1</sup> Quién sabe cuánto tiempo seguirán funcionando nuestras Mac anteriores a Intel, así que si alguien de Apple está escuchando: cree una versión compatible con OS X de ¡MacDraw Pro!

### Agradecimientos a la tercera edición

Hemos estado trabajando con MIT Press durante más de dos décadas, ¡y qué excelente relación ha sido! Agradecemos a Ellen Faran, Bob Prior, Ada Brunstein y Mary Reilly por su ayuda y apoyo.

Estábamos distribuidos geográficamente mientras producíamos la tercera edición, trabajando en el Departamento de Ciencias de la Computación de Dartmouth College, el Departamento de Computación del MIT

<sup>1</sup>Investigamos varios programas de dibujo que se ejecutan en Mac OS X, pero todos tenían deficiencias significativas en comparación con MacDraw Pro. Intentamos brevemente producir las ilustraciones para este libro con un programa de dibujo diferente y bien conocido. Descubrimos que tomaba al menos cinco veces más tiempo producir cada ilustración que con MacDraw Pro, y las ilustraciones resultantes no se veían tan bien. De ahí la decisión de volver a MacDraw Pro ejecutándose en Macintosh más antiguos.

Laboratorio de Ciencia e Inteligencia Artificial, y el Departamento de Ingeniería Industrial e Investigación de Operaciones de la Universidad de Columbia. Agradecemos a nuestras respectivas universidades y colegas por proporcionar entornos tan estimulantes y de apoyo.

Julie Sussman, PPA, una vez más nos rescató como correctora técnica. Una y otra vez nos asombraban los errores que se nos escapaban, pero que Julie captaba. También nos ayudó a mejorar nuestra presentación en varios lugares. Si hay un Salón de la Fama para los correctores de estilo técnicos, Julie es una miembro infalible en la primera votación. Ella es nada menos que fenomenal. ¡Gracias, gracias, gracias, Julio! Priya Natarajan también encontró algunos errores que pudimos corregir antes de que este libro saliera a la imprenta. Cualquier error que quede (y, sin duda, algunos quedan) es responsabilidad de los autores (y probablemente se insertaron después de que Julie leyera el material).

El tratamiento de los árboles de van Emde Boas se deriva de las notas de Erik Demaine, que a su vez fueron influenciadas por Michael Bender. También incorporamos ideas de Javed Aslam, Bradley Kuszmaul y Hui Zha en esta edición.

El capítulo sobre subprocessos múltiples se basó en notas escritas originalmente en conjunto con Harald Prokop. El material fue influenciado por varios otros que trabajaban en el proyecto Cilk en el MIT, incluidos Bradley Kuszmaul y Matteo Frigo. El diseño del pseudocódigo multiproceso se inspiró en las extensiones Cilk del MIT para C y en las extensiones Cilk++ de Cilk Arts para C++.

También agradecemos a los numerosos lectores de la primera y segunda ediciones que informaron errores o enviaron sugerencias sobre cómo mejorar este libro. Corregimos todos los errores de buena fe que se informaron e incorporamos tantas sugerencias como pudimos. Nos regocijamos de que el número de tales contribuyentes haya crecido tanto que debemos lamentar que se haya vuelto poco práctico enumerarlos a todos.

Finalmente, agradecemos a nuestras esposas, Nicole Cormen, Wendy Leiserson, Gail Rivest y Rebecca Ivry, y a nuestros hijos, Ricky, Will, Debby y Katie Leiserson; Alex y Christopher Rivest; y Molly, Noah y Benjamin Stein, por su amor y apoyo mientras preparábamos este libro. La paciencia y el ánimo de nuestras familias hicieron posible este proyecto. Les dedicamos con cariño este libro.

TOMÁS H. CORMEN

Líbano, Nuevo Hampshire

CHARLES E. LEISERSON

Cambridge, Massachusetts

RONALD L. RIVEST

Cambridge, Massachusetts

CLIFFORD STEIN

Nueva York, Nueva York

febrero de 2009



---

# Introducción a los Algoritmos

Tercera edición

---

## Fundaciones

---

## Introducción

Esta parte lo iniciará a pensar en diseñar y analizar algoritmos. Pretende ser una breve introducción a cómo especificamos los algoritmos, algunas de las estrategias de diseño que usaremos a lo largo de este libro y muchas de las ideas fundamentales que se usan en el análisis de algoritmos. Las partes posteriores de este libro se basarán en esta base.

El capítulo 1 proporciona una descripción general de los algoritmos y su lugar en los sistemas informáticos modernos. Este capítulo define qué es un algoritmo y enumera algunos ejemplos. También argumenta que deberíamos considerar los algoritmos como una tecnología, junto con tecnologías como hardware rápido, interfaces gráficas de usuario, sistemas orientados a objetos y redes.

En el Capítulo 2, vemos nuestros primeros algoritmos, que resuelven el problema de ordenar una secuencia de  $n$  números. Están escritos en un pseudocódigo que, aunque no se puede traducir directamente a ningún lenguaje de programación convencional, transmite la estructura del algoritmo con la suficiente claridad como para poder implementarlo en el lenguaje de su elección. Los algoritmos de clasificación que examinamos son la clasificación por inserción, que utiliza un enfoque incremental, y la clasificación por fusión, que utiliza una técnica recursiva conocida como "divide y vencerás". Aunque el tiempo que cada uno requiere aumenta con el valor de  $n$ , la tasa de aumento difiere entre los dos algoritmos. Determinaremos estos tiempos de ejecución en el Capítulo 2 y desarrollaremos una notación útil para expresarlos.

El capítulo 3 define con precisión esta notación, a la que llamamos notación asintótica. Comienza definiendo varias notaciones asintóticas, que usamos para delimitar los tiempos de ejecución del algoritmo desde arriba y/o desde abajo. El resto del Capítulo 3 es principalmente una presentación de la notación matemática, más para asegurarse de que su uso de la notación coincida con el de este libro que para enseñarle nuevos conceptos matemáticos.

El Capítulo 4 profundiza más en el método de divide y vencerás presentado en el Capítulo 2. Proporciona ejemplos adicionales de algoritmos de divide y vencerás, incluido el sorprendente método de Strassen para multiplicar dos matrices cuadradas. El capítulo 4 contiene métodos para resolver recurrencias, que son útiles para describir los tiempos de ejecución de los algoritmos recursivos. Una técnica poderosa es el "método maestro", que a menudo usamos para resolver las recurrencias que surgen de los algoritmos de divide y vencerás. Aunque gran parte del Capítulo 4 está dedicado a probar la corrección del método maestro, puede omitir esta prueba y seguir empleando el método maestro.

El Capítulo 5 presenta el análisis probabilístico y los algoritmos aleatorios. Por lo general, usamos el análisis probabilístico para determinar el tiempo de ejecución de un algoritmo en los casos en los que, debido a la presencia de una distribución de probabilidad inherente, el tiempo de ejecución puede diferir en diferentes entradas del mismo tamaño. En algunos casos, asumimos que las entradas se ajustan a una distribución de probabilidad conocida, de modo que estamos promediando el tiempo de ejecución de todas las entradas posibles. En otros casos, la distribución de probabilidad no proviene de las entradas sino de elecciones aleatorias realizadas durante el curso del algoritmo. Un algoritmo cuyo comportamiento está determinado no solo por su entrada sino también por los valores producidos por un generador de números aleatorios es un algoritmo aleatorio. Podemos usar algoritmos aleatorios para imponer una distribución de probabilidad en las entradas, asegurando así que ninguna entrada en particular siempre cause un rendimiento deficiente, o incluso limitar la tasa de error de los algoritmos que pueden producir resultados incorrectos de forma limitada.

Los apéndices A–D contienen otro material matemático que le resultará útil a medida que lea este libro. Es probable que haya visto gran parte del material de los capítulos del apéndice antes de leer este libro (aunque las definiciones específicas y las convenciones de notación que usamos pueden diferir en algunos casos de lo que ha visto en el pasado), por lo que debe pensar en los Apéndices como material de referencia.

Por otro lado, es probable que aún no haya visto la mayor parte del material de la Parte I. Todos los capítulos de la Parte I y los Apéndices están escritos con un estilo tutorial.

---

# 1

# El papel de los algoritmos en la computación

¿Qué son los algoritmos? ¿Por qué vale la pena el estudio de los algoritmos? ¿Cuál es el papel de los algoritmos en relación con otras tecnologías utilizadas en las computadoras? En este capítulo responderemos a estas preguntas.

---

## 1.1 Algoritmos

Informalmente, un algoritmo es cualquier procedimiento computacional bien definido que toma algún valor, o conjunto de valores, como entrada y produce algún valor, o conjunto de valores, como salida. Un algoritmo es, por lo tanto, una secuencia de pasos computacionales que transforman la entrada en la salida.

También podemos ver un algoritmo como una herramienta para resolver un problema computacional bien especificado. El enunciado del problema especifica en términos generales la relación entrada/salida deseada. El algoritmo describe un procedimiento computacional específico para lograr esa relación de entrada/salida.

Por ejemplo, podríamos necesitar clasificar una secuencia de números en orden no decreciente. Este problema surge con frecuencia en la práctica y proporciona un terreno fértil para la introducción de muchas técnicas de diseño estándar y herramientas de análisis. Así es como definimos formalmente el problema de clasificación:

Entrada: Una secuencia de  $n$  números  $a_1; a_2; \dots; a_n$ .

Salida: Una permutación (reordenación)  $a_0$  que  $a_1; a_2; \dots; a_n$  ni de la secuencia de entrada tal que  $a_0 < a_1 < a_2 < \dots < a_{n-1} < a_n$ .

Por ejemplo, dada la secuencia de entrada  $h31; 41; 59; 26; 41; 58$ , un algoritmo de clasificación devuelve como salida la secuencia  $h26; 31; 41; 41; 58; 59$ . Tal secuencia de entrada se llama una instancia del problema de clasificación. En general, una instancia de un problema consiste en la entrada (que satisface las restricciones impuestas en el enunciado del problema) necesaria para calcular una solución al problema.

Debido a que muchos programas lo utilizan como un paso intermedio, la clasificación es una operación fundamental en informática. Como resultado, tenemos una gran cantidad de buenos algoritmos de clasificación a nuestra disposición. Qué algoritmo es mejor para una aplicación determinada depende, entre otros factores, de la cantidad de elementos que se ordenarán, la medida en que los elementos ya están algo ordenados, las posibles restricciones en los valores de los elementos, la arquitectura de la computadora y el tipo. de dispositivos de almacenamiento a utilizar: memoria principal, discos o incluso cintas.

Se dice que un algoritmo es correcto si, para cada instancia de entrada, se detiene con la salida correcta. Decimos que un algoritmo correcto resuelve el problema computacional dado. Es posible que un algoritmo incorrecto no se detenga en absoluto en algunas instancias de entrada, o que se detenga con una respuesta incorrecta. Al contrario de lo que cabría esperar, los algoritmos incorrectos a veces pueden ser útiles, si podemos controlar su tasa de error. Veremos un ejemplo de un algoritmo con una tasa de error controlable en el capítulo 31 cuando estudiemos algoritmos para encontrar números primos grandes. Normalmente, sin embargo, nos ocuparemos sólo de los algoritmos correctos.

Un algoritmo se puede especificar en inglés, como un programa de computadora o incluso como un diseño de hardware. El único requisito es que la especificación proporcione una descripción precisa del procedimiento de cálculo a seguir.

### ¿Qué tipo de problemas se resuelven mediante algoritmos?

La clasificación no es de ninguna manera el único problema computacional para el cual se han desarrollado algoritmos. (Probablemente lo sospechaste cuando viste el tamaño de este libro). Las aplicaciones prácticas de los algoritmos son ubicuas e incluyen los siguientes ejemplos:

El Proyecto Genoma Humano ha hecho un gran progreso hacia las metas de identificar los 100 000 genes en el ADN humano, determinar las secuencias de los 3 000 millones de pares de bases químicas que componen el ADN humano, almacenar esta información en bases de datos y desarrollar herramientas para datos. análisis. Cada uno de estos pasos requiere algoritmos sofisticados. Aunque las soluciones a los diversos problemas involucrados están más allá del alcance de este libro, muchos métodos para resolver estos problemas biológicos usan ideas de varios de los capítulos de este libro, lo que permite a los científicos realizar tareas mientras usan los recursos de manera eficiente. Los ahorros son en tiempo, tanto humano como de máquinas, y en dinero, ya que se puede extraer más información de las técnicas de laboratorio.

Internet permite a las personas de todo el mundo acceder y recuperar rápidamente grandes cantidades de información. Con la ayuda de algoritmos inteligentes, los sitios en Internet pueden administrar y manipular este gran volumen de datos. Los ejemplos de problemas que hacen un uso esencial de los algoritmos incluyen encontrar buenas rutas por las que viajarán los datos (las técnicas para resolver tales problemas aparecen en

## 1.1 Algoritmos

7

Capítulo 24), y usar un motor de búsqueda para encontrar rápidamente páginas en las que reside información particular (las técnicas relacionadas se encuentran en los Capítulos 11 y 32).

El comercio electrónico permite negociar e intercambiar electrónicamente bienes y servicios, y depende de la privacidad de la información personal, como números de tarjetas de crédito, contraseñas y extractos bancarios. Las tecnologías centrales que se utilizan en el comercio electrónico incluyen la criptografía de clave pública y las firmas digitales (tratadas en el Capítulo 31), que se basan en algoritmos numéricos y teoría de números.

Las empresas manufactureras y otras empresas comerciales a menudo necesitan asignar recursos escasos de la manera más beneficiosa. Una compañía petrolera puede desear saber dónde colocar sus pozos para maximizar su beneficio esperado. Un candidato político puede querer determinar dónde gastar dinero comprando publicidad de campaña para maximizar las posibilidades de ganar una elección. Una línea aérea puede desear asignar tripulaciones a los vuelos de la manera menos costosa posible, asegurándose de que se cubra cada vuelo y de que se cumplan las normas gubernamentales relativas a la programación de la tripulación. Un proveedor de servicios de Internet puede desear determinar dónde colocar recursos adicionales para servir a sus clientes de manera más eficaz. Todos estos son ejemplos de problemas que pueden resolverse mediante programación lineal, que estudiaremos en el capítulo 29.

Aunque algunos de los detalles de estos ejemplos están más allá del alcance de este libro, brindamos técnicas subyacentes que se aplican a estos problemas y áreas problemáticas. También mostramos cómo resolver muchos problemas específicos, incluidos los siguientes:

Nos dan un mapa de carreteras en el que está marcada la distancia entre cada par de intersecciones adyacentes, y deseamos determinar la ruta más corta de una intersección a otra. La cantidad de rutas posibles puede ser enorme, incluso si no permitimos rutas que se cruzan entre sí. ¿Cómo elegimos cuál de todos

posibles rutas es la más corta? Aquí, modelamos el mapa de carreteras (que en sí mismo es un modelo de las carreteras reales) como un gráfico (que veremos en la Parte VI y el Apéndice B), y deseamos encontrar el camino más corto de un vértice a otro en el gráfico. . Veremos cómo resolver este problema de manera eficiente en el Capítulo 24.

Tenemos dos secuencias ordenadas de símbolos, XD hx1; x2;:::xmi y YD hy1; y2;:::yni, y deseamos encontrar una subsecuencia común más larga de X y una subsecuencia YA de X es solo X con algunos (o posiblemente todos o ninguno) de sus elementos eliminados. Por ejemplo, una subsecuencia de hA; B; C; D; MI; F; Gi sería hB; C; MI; Soldado americano. La longitud de una subsecuencia común más larga de X e Y da una medida de cuán similares son estas dos secuencias. Por ejemplo, si las dos secuencias son pares de bases en hebras de ADN, podríamos considerarlas similares si tienen una subsecuencia común larga. Si X tiene m símbolos e Y tiene n símbolos, entonces X e Y tienen  $2^m$  y  $2^n$  posibles subsecuencias,

respectivamente. Seleccionar todas las subsecuencias posibles de X e Y y unirlas podría tomar un tiempo prohibitivamente largo a menos que m y n sean muy pequeños.

Veremos en el Capítulo 15 cómo usar una técnica general conocida como programación dinámica para resolver este problema de manera mucho más eficiente.

Tenemos un diseño mecánico en términos de una biblioteca de partes, donde cada parte puede incluir instancias de otras partes, y necesitamos enumerar las partes en orden para que cada parte aparezca antes que cualquier parte que la use. Si el diseño consta de n partes, entonces hay  $n!$  órdenes posibles, donde  $n!$  denota la función factorial.

Debido a que la función factorial crece más rápido que incluso una función exponencial, no podemos generar de manera factible cada orden posible y luego verificar que, dentro de ese orden, cada parte aparece antes que las partes que la usan (a menos que solo tengamos unas pocas partes). Este problema es una instancia de clasificación topológica y en el Capítulo 22 veremos cómo resolver este problema de manera eficiente.

Nos dan n puntos en el plano, y deseamos encontrar la envolvente convexa de estos puntos. El casco convexo es el polígono convexo más pequeño que contiene los puntos.

Intuitivamente, podemos pensar que cada punto está representado por un clavo que sobresale de una tabla. El casco convexo estaría representado por una tira elástica apretada que rodea todos los clavos. Cada clavo alrededor del cual gira la banda de goma es un vértice del casco convexo. (Consulte la Figura 33.6 en la página 1029 para ver un ejemplo). Cualquiera de los  $2^n$  subconjuntos de los puntos podría ser los vértices del casco convexo. Saber qué puntos son vértices del casco convexo tampoco es suficiente, ya que también necesitamos saber el orden en que aparecen. Hay muchas opciones, por lo tanto, para los vértices del casco convexo.

El capítulo 33 da dos buenos métodos para encontrar el casco convexo.

Estas listas están lejos de ser exhaustivas (como probablemente habrá deducido nuevamente por el peso de este libro), pero exhiben dos características que son comunes a muchos problemas algorítmicos interesantes:

1. Tienen muchas soluciones candidatas, la gran mayoría de las cuales no resuelven el problema en cuestión. Encontrar uno que lo haga, o uno que sea "mejor", puede presentar todo un desafío.
2. Tienen aplicaciones prácticas. De los problemas de la lista anterior, encontrar la ruta más corta proporciona los ejemplos más sencillos. Una empresa de transporte, como una empresa de camiones o ferrocarriles, tiene un interés financiero en encontrar los caminos más cortos a través de una red vial o ferroviaria porque tomar caminos más cortos da como resultado menores costos de mano de obra y combustible. O un nodo de enrutamiento en Internet puede necesitar encontrar la ruta más corta a través de la red para enrutar un mensaje rápidamente. O una persona que desee conducir de Nueva York a Boston puede querer encontrar direcciones de manejo en un sitio web apropiado, o puede usar su GPS mientras conduce.

No todos los problemas resueltos por algoritmos tienen un conjunto de soluciones candidatas fácilmente identificables. Por ejemplo, supongamos que se nos da un conjunto de valores numéricos que representan muestras de una señal y queremos calcular la transformada discreta de Fourier de estas muestras. La transformada discreta de Fourier convierte el dominio del tiempo al dominio de la frecuencia, produciendo un conjunto de coeficientes numéricos, de modo que podamos determinar la fuerza de varias frecuencias en la señal muestreada. Además de estar en el centro del procesamiento de señales, las transformadas discretas de Fourier tienen aplicaciones en la compresión de datos y en la multiplicación de polinomios y números enteros grandes. El capítulo 30 proporciona un algoritmo eficiente, la transformada rápida de Fourier (comúnmente llamada FFT), para este problema, y el capítulo también esboza el diseño de un circuito de hardware para calcular la FFT.

### Estructuras de datos

Este libro también contiene varias estructuras de datos. Una estructura de datos es una forma de almacenar y organizar datos para facilitar el acceso y las modificaciones. Ninguna estructura de datos única funciona bien para todos los propósitos, por lo que es importante conocer las fortalezas y limitaciones de varias de ellas.

### Técnica

Aunque puede usar este libro como un "libro de recetas" para algoritmos, es posible que algún día encuentre un problema para el cual no pueda encontrar fácilmente un algoritmo publicado (muchos de los ejercicios y problemas de este libro, por ejemplo). Este libro le enseñará técnicas de diseño y análisis de algoritmos para que pueda desarrollar algoritmos por su cuenta, demostrar que dan la respuesta correcta y comprender su eficiencia.

Diferentes capítulos abordan diferentes aspectos de la resolución de problemas algorítmicos. Algunos capítulos abordan problemas específicos, como encontrar medianas y ordenar estadísticas en el Capítulo 9, calcular árboles de expansión mínimos en el Capítulo 23 y determinar un flujo máximo en una red en el Capítulo 26. Otros capítulos abordan técnicas, como divide y vencerás en Capítulo 4, programación dinámica en el Capítulo 15 y análisis amortizado en el Capítulo 17.

### problemas difíciles

La mayor parte de este libro trata sobre algoritmos eficientes. Nuestra medida habitual de eficiencia es la velocidad, es decir, cuánto tarda un algoritmo en producir su resultado. Sin embargo, existen algunos problemas para los que no se conoce una solución eficaz. El Capítulo 34 estudia un subconjunto interesante de estos problemas, que se conocen como NP-completos.

¿Por qué son interesantes los problemas NP-completos? Primero, aunque nunca se ha encontrado un algoritmo eficiente para un problema NP-completo, nadie ha probado

que no puede existir un algoritmo eficiente para uno. En otras palabras, nadie sabe si existen o no algoritmos eficientes para problemas NP-completos. Segundo, el conjunto de problemas NP-completos tiene la notable propiedad de que si existe un algoritmo eficiente para cualquiera de ellos, entonces existen algoritmos eficientes para todos ellos. Esta relación entre los problemas NP-completos hace que la falta de soluciones eficientes sea aún más tentadora. En tercer lugar, varios problemas NP-completos son similares, pero no idénticos, a problemas para los que conocemos algoritmos eficientes. Los informáticos están intrigados por cómo un pequeño cambio en la declaración del problema puede causar un gran cambio en la eficiencia del algoritmo más conocido.

Debe conocer los problemas NP-completos porque algunos de ellos surgen con una frecuencia sorprendente en aplicaciones reales. Si se le pide que produzca un algoritmo eficiente para un problema NP-completo, es probable que pase mucho tiempo en una búsqueda infructuosa. Si puede demostrar que el problema es NP-completo, puede dedicar su tiempo a desarrollar un algoritmo eficiente que brinde una buena solución, pero no la mejor posible.

Como ejemplo concreto, considere una empresa de entrega con un depósito central. Cada día, carga cada camión de reparto en el depósito y lo envía para entregar mercancías a varias direcciones. Al final del día, cada camión debe regresar al depósito para que esté listo para ser cargado al día siguiente. Para reducir los costos, la empresa desea seleccionar un orden de paradas de entrega que produzca la menor distancia total recorrida por cada camión. Este problema es el conocido "problema del viajante de comercio" y es NP-completo. No tiene un algoritmo eficiente conocido. Bajo ciertas suposiciones, sin embargo, conocemos algoritmos eficientes que dan una distancia total que no está muy por encima de la más pequeña posible. El Capítulo 35 analiza tales "algoritmos de aproximación".

### Paralelismo

Durante muchos años, podíamos contar con que las velocidades de reloj del procesador aumentaran a un ritmo constante. Sin embargo, las limitaciones físicas presentan un obstáculo fundamental para las velocidades de reloj cada vez mayores: debido a que la densidad de potencia aumenta superlinealmente con la velocidad del reloj, los chips corren el riesgo de derretirse una vez que sus velocidades de reloj sean lo suficientemente altas. Por lo tanto, para realizar más cálculos por segundo, los chips se están diseñando para contener no solo uno, sino varios "núcleos" de procesamiento. Podemos comparar estas computadoras multinúcleo con varias computadoras secuenciales en un solo chip; en otras palabras, son un tipo de "computadora paralela". Para obtener el mejor rendimiento de las computadoras multinúcleo, debemos diseñar algoritmos teniendo en cuenta el paralelismo. El Capítulo 27 presenta un modelo para algoritmos "multiproceso", que aprovechan múltiples núcleos. Este modelo tiene ventajas desde un punto de vista teórico y forma la base de varios programas informáticos exitosos, incluido un camp

**Ejercicios****1.1-1**

Dé un ejemplo del mundo real que requiera clasificación o un ejemplo del mundo real que requiera calcular un casco convexo.

**1.1-2**

Además de la velocidad, ¿qué otras medidas de eficiencia se podrían usar en un entorno del mundo real?

**1.1-3**

Seleccione una estructura de datos que haya visto anteriormente y discuta sus fortalezas y limitaciones.

**1.1-4**

¿En qué se parecen los problemas del camino más corto y del viajante de comercio dados anteriormente?  
¿En qué se diferencian?

**1.1-5**

Piense en un problema del mundo real en el que solo sirva la mejor solución. Luego piense en una en la que una solución que sea "aproximadamente" la mejor sea lo suficientemente buena.

---

## 1.2 Algoritmos como tecnología

Supongamos que las computadoras fueran infinitamente rápidas y que la memoria de la computadora fuera gratuita. ¿Tendrías alguna razón para estudiar algoritmos? La respuesta es sí, aunque solo sea porque le gustaría demostrar que su método de solución termina y lo hace con la respuesta correcta.

Si las computadoras fueran infinitamente rápidas, cualquier método correcto para resolver un problema serviría. Probablemente desee que su implementación esté dentro de los límites de las buenas prácticas de ingeniería de software (por ejemplo, su implementación debe estar bien diseñada y documentada), pero con mayor frecuencia usará el método que sea más fácil de implementar.

Por supuesto, las computadoras pueden ser rápidas, pero no son infinitamente rápidas. Y la memoria puede ser económica, pero no es gratuita. Por lo tanto, el tiempo de computación es un recurso limitado, al igual que el espacio en la memoria. Debe usar estos recursos con prudencia, y los algoritmos que son eficientes en términos de tiempo o espacio lo ayudarán a hacerlo.

### Eficiencia

Diferentes algoritmos ideados para resolver el mismo problema a menudo difieren dramáticamente en su eficiencia. Estas diferencias pueden ser mucho más importantes que las diferencias debidas al hardware y al software.

Como ejemplo, en el Capítulo 2, veremos dos algoritmos para ordenar. La primera, conocida como ordenación por inserción, toma un tiempo aproximadamente igual a  $c_1n^2$  para ordenar  $n$  elementos, donde  $c_1$  es una constante que no depende de  $n$ . Es decir, lleva un tiempo aproximadamente proporcional a  $n^2$ . El segundo, ordenar por fusión, toma un tiempo aproximadamente igual a  $c_2n \lg n$ , donde  $\lg n$  representa  $\log_2 n$  y  $c_2$  es otra constante que tampoco depende de  $n$ . La ordenación por inserción normalmente tiene un factor constante más pequeño que la ordenación por fusión, de modo que  $c_1 < c_2$ . Veremos que los factores constantes pueden tener mucho menos impacto en el tiempo de ejecución que la dependencia del tamaño de entrada  $n$ . Escribamos el tiempo de ejecución de la ordenación por inserción como  $c_1n$  y fusionemos el tiempo de ejecución de la ordenación como  $c_2n \lg n$ . Entonces vemos que donde la ordenación por inserción tiene un factor de  $n$  en su tiempo de ejecución, la ordenación por fusión tiene un factor de  $\lg n$ , que es mucho menor. (Por ejemplo, cuando  $n = 1000$ ,  $\lg n$  es aproximadamente 10, y cuando  $n$  es igual a un millón,  $\lg n$  es aproximadamente solo 20). Lo suficientemente grande, la ventaja de  $\lg n$  vs.  $n$  de merge sort compensará con creces la diferencia en factores constantes. No importa cuánto más pequeño sea  $c_1$  que  $c_2$ , siempre habrá un punto de cruce más allá del cual la ordenación por combinación es más rápida.

Para un ejemplo concreto, enfrentemos una computadora más rápida (computadora A) que ejecuta ordenación por inserción contra una computadora más lenta (computadora B) que ejecuta ordenación por fusión. Cada uno debe ordenar una matriz de 10 millones de números. (Aunque 10 millones de números pueden parecer muchos, si los números son enteros de ocho bytes, entonces la entrada ocupa alrededor de 80 megabytes, lo que cabe en la memoria incluso de una computadora portátil económica muchas veces). Suponga que la computadora A ejecuta 10 mil millones de instrucciones por segundo (más rápido que cualquier computadora secuencial individual en el momento de escribir este artículo) y la computadora B ejecuta solo 10 millones de instrucciones por segundo, por lo que la computadora A es 1000 veces más rápida que la computadora B en potencia de cálculo bruta. Para hacer la diferencia aún más dramática, supongamos que el programador más astuto del mundo codifica la ordenación por inserción en lenguaje de máquina para la computadora A, y el código resultante requiere  $2n^2$  instrucciones para ordenar  $n$  números. Supongamos además que solo un programador promedio implementa la ordenación por fusión, usando un lenguaje de alto nivel con un compilador ineficiente, con el código resultante tomando  $50n \lg n$  instrucciones. Para ordenar 10 millones de números, la computadora A toma

$$\frac{2 \cdot 1072 \text{ instrucciones}}{\text{segundo}} D = 20.000 \text{ segundos (más de 5,5 horas)} ; 1010 \text{ instrucciones/}$$

mientras que la computadora B toma

$$\frac{50 \cdot 107 \lg 107 \text{ instrucciones}}{\text{instrucciones/segundo}} = 1163 \text{ segundos (menos de 20 minutos)}$$

Al usar un algoritmo cuyo tiempo de ejecución crece más lentamente, incluso con un compilador deficiente, ¡la computadora B funciona más de 17 veces más rápido que la computadora A! La ventaja de la ordenación por fusión es aún más pronunciada cuando ordenamos 100 millones de números: mientras que la ordenación por inserción lleva más de 23 días, la ordenación por fusión tarda menos de cuatro horas. En general, a medida que aumenta el tamaño del problema, también lo hace la ventaja relativa de la ordenación por fusión.

### Algoritmos y otras tecnologías

El ejemplo anterior muestra que debemos considerar los algoritmos, como el hardware de una computadora, como una tecnología. El rendimiento total del sistema depende tanto de elegir algoritmos eficientes como de elegir hardware rápido. Así como se están logrando rápidos avances en otras tecnologías informáticas, también se están logrando en algoritmos.

Quizás se pregunte si los algoritmos son realmente tan importantes en las computadoras contemporáneas a la luz de otras tecnologías avanzadas, como

- arquitecturas informáticas avanzadas y tecnologías de fabricación,
- interfaces gráficas de usuario (GUI) intuitivas y fáciles de usar,
- sistemas orientados a objetos,
- tecnologías web integradas y redes
- rápidas, tanto cableadas como inalámbricas.

La respuesta es sí. Aunque algunas aplicaciones no requieren explícitamente contenido de micrófono algorítmico en el nivel de la aplicación (como algunas aplicaciones simples basadas en la Web), muchas sí lo requieren. Por ejemplo, considere un servicio basado en la web que determina cómo viajar de un lugar a otro. Su implementación se basaría en un hardware rápido, una interfaz gráfica de usuario, redes de área amplia y también, posiblemente, en la orientación a objetos. Sin embargo, también requeriría algoritmos para ciertas operaciones, como encontrar rutas (probablemente usando un algoritmo de ruta más corta), renderizar mapas e interpolar direcciones.

Además, incluso una aplicación que no requiere contenido algorítmico a nivel de aplicación depende en gran medida de los algoritmos. ¿La aplicación depende de un hardware rápido? El diseño del hardware utilizó algoritmos. ¿La aplicación se basa en interfaces gráficas de usuario? El diseño de cualquier GUI se basa en algoritmos. ¿La aplicación se basa en redes? El enruteamiento en las redes depende en gran medida de los algoritmos.

¿La aplicación estaba escrita en un lenguaje diferente al código de máquina? Luego fue procesado por un compilador, intérprete o ensamblador, todos los cuales hacen un uso extensivo

de algoritmos. Los algoritmos son el núcleo de la mayoría de las tecnologías utilizadas en las computadoras contemporáneas.

Además, con las capacidades cada vez mayores de las computadoras, las usamos para resolver problemas más grandes que nunca. Como vimos en la comparación anterior entre la ordenación por inserción y la ordenación por fusión, es en problemas de mayor tamaño que las diferencias en eficiencia entre algoritmos se vuelven particularmente prominentes.

Tener una base sólida de conocimientos y técnicas algorítmicas es una característica que separa a los programadores verdaderamente hábiles de los novatos. Con la tecnología informática moderna, puede realizar algunas tareas sin saber mucho sobre algoritmos, pero con una buena experiencia en algoritmos, puede hacer mucho, mucho. más.

### Ejercicios

#### 1.2-1

Dé un ejemplo de una aplicación que requiera contenido algorítmico en el nivel de aplicación y discuta la función de los algoritmos involucrados.

#### 1.2-2

Suponga que estamos comparando implementaciones de clasificación por inserción y clasificación por fusión en la misma máquina. Para entradas de tamaño  $n$ , la ordenación por inserción se ejecuta en pasos de  $8n^2$ , mientras que la ordenación por fusión se ejecuta en pasos de  $64n \lg n$ . ¿Para qué valores de  $n$  la ordenación por inserción supera a la ordenación por fusión?

#### 1.2-3

¿Cuál es el valor más pequeño de  $n$  tal que un algoritmo cuyo tiempo de ejecución es  $100n^2$  se ejecuta más rápido que un algoritmo cuyo tiempo de ejecución es  $2n$  en la misma máquina?

---

## Problemas

1-1 Comparación de tiempos de ejecución Para cada función  $f(n)$  y el tiempo  $t$  en la siguiente tabla, determine el mayor tamaño  $n$  de un problema que se puede resolver en el tiempo  $t$ , suponiendo que el algoritmo para resolver el problema toma  $f(n) / \text{microsegundos}$ .

	1 1 segundo	1 minuto hora	1	1 día	1 mes	1 año	1 siglo
$\lg n$							
$\frac{1}{pn}$							
$n^{\text{constante}}$							
$n \lg n$							
$n^2$							
$n^3$							
$2^n$							
$n^{\frac{1}{2}}$							

---

### Notas del capítulo

Hay muchos textos excelentes sobre el tema general de los algoritmos, incluidos los de Aho, Hopcroft y Ullman [5, 6]; Baase y Van Gelder [28]; Brassard y Bratley [54]; Dasgupta, Papadimitriou y Vazirani [82]; Goodrich y Tamassia [148]; Hofri [175]; Horowitz, Sahni y Rajasekaran [181]; Johnsonbaugh y Schaefer [193]; Kingston [205]; Kleinberg y Tardos [208]; Knuth [209, 210, 211]; Kozen [220]; Levitina [235]; Manber [242]; Mehlhorn [249, 250, 251]; Purdom y Brown [287]; Reingold, Nievergelt y Deo [293]; Sedgewick [306]; Sedgewick y Flajolet [307]; Skiena [318]; y Wilf [356]. Bentley [42, 43] y Gonnet [145] analizan algunos de los aspectos más prácticos del diseño de algoritmos. Las encuestas del campo de los algoritmos también se pueden encontrar en el Handbook of Theoretical Computer Science, Volumen A [342] y el CRC Algorithms and Theory of Computation Handbook [25]. Se pueden encontrar descripciones generales de los algoritmos utilizados en biología computacional en los libros de texto de Gusfield [156], Pevzner [275], Setubal y Meidanis [310] y Waterman [350].

## 2 Empezando

Este capítulo lo familiarizará con el marco que usaremos a lo largo del libro para pensar en el diseño y análisis de algoritmos. Es independiente, pero incluye varias referencias al material que presentamos en los Capítulos 3 y 4.

(También contiene varias sumas, que el Apéndice A muestra cómo resolver).

Comenzamos examinando el algoritmo de clasificación por inserción para resolver el problema de clasificación presentado en el Capítulo 1. Definimos un "pseudocódigo" que debería resultarle familiar si ha hecho programación de computadoras, y lo usamos para mostrar cómo especificaremos nuestros algoritmos. Habiendo especificado el algoritmo de ordenación por inserción, argumentamos que ordena correctamente y analizamos su tiempo de ejecución. El análisis introduce una notación que se enfoca en cómo ese tiempo aumenta con la cantidad de elementos a clasificar.

Después de nuestra discusión sobre la ordenación por inserción, presentamos el enfoque de divide y vencerás para el diseño de algoritmos y lo usamos para desarrollar un algoritmo llamado ordenación por fusión.

Terminamos con un análisis del tiempo de ejecución de merge sort.

### 2.1 Clasificación por inserción

Nuestro primer algoritmo, clasificación por inserción, resuelve el problema de clasificación presentado en el Capítulo 1:

Entrada: Una secuencia de  $n$  números  $a_1; a_2; \dots; a_n$ .

Salida: Una permutación (reordenación)  $a'_0$  que  $a'_0 = a_1; a'_0 = a_2; \dots; a'_0 = a_n$  ni de la secuencia de entrada tal

$$1 \quad a'_0 \quad a'_{n-1}$$

Los números que deseamos ordenar también se conocen como claves. Aunque conceptualmente estamos ordenando una secuencia, la entrada nos llega en forma de una matriz con  $n$  elementos.

En este libro, normalmente describiremos los algoritmos como programas escritos en un pseudocódigo que es similar en muchos aspectos a C, C++, Java, Python o Pascal. Si ha sido introducido a alguno de estos idiomas, no debería tener problemas

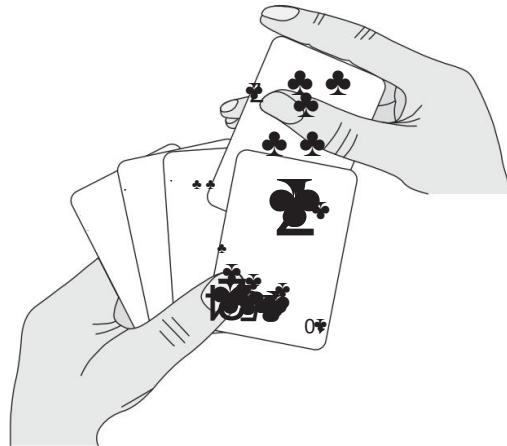


Figura 2.1 Clasificación de una mano de cartas mediante clasificación por inserción.

leyendo nuestros algoritmos. Lo que separa al pseudocódigo del código "real" es que en el pseudocódigo, empleamos cualquier método expresivo que sea más claro y conciso para especificar un algoritmo dado. A veces, el método más claro es el inglés, así que no se sorprenda si encuentra una frase u oración en inglés incrustada en una sección de código "real". Otra diferencia entre el pseudocódigo y el código real es que el pseudocódigo normalmente no se ocupa de cuestiones de ingeniería de software.

Los problemas de abstracción de datos, modularidad y manejo de errores a menudo se ignoran para transmitir la esencia del algoritmo de manera más concisa.

Comenzamos con la ordenación por inserción, que es un algoritmo eficiente para ordenar una pequeña cantidad de elementos. La clasificación por inserción funciona de la misma manera que muchas personas clasifican una mano de naipes. Empezamos con la mano izquierda vacía y las cartas boca abajo sobre la mesa. Luego retiramos una carta a la vez de la mesa y la insertamos en la posición correcta en la mano izquierda. Para encontrar la posición correcta de una carta, la comparamos con cada una de las cartas que ya están en la mano, de derecha a izquierda, como se ilustra en la Figura 2.1. En todo momento, se ordenan las cartas que se sostienen en la mano izquierda, y estas cartas eran originalmente las primeras cartas de la pila sobre la mesa.

Presentamos nuestro pseudocódigo para ordenar por inserción como un procedimiento llamado CLASIFICACIÓN POR INSERCIÓN , que toma como parámetro un arreglo  $A[1 : n]$  que contiene una secuencia de longitud  $n$  que se va a ordenar. (En el código, el número  $n$  de elementos en  $A$  se denota por  $A$ : longitud). El algoritmo ordena los números de entrada en su lugar: reorganiza los números dentro de la matriz  $A$ , con un número constante de ellos almacenados fuera de la matriz como máximo. en cualquier momento. La matriz de entrada  $A$  contiene la secuencia de salida ordenada cuando finaliza el procedimiento INSERCIÓN-ORDENACIÓN .

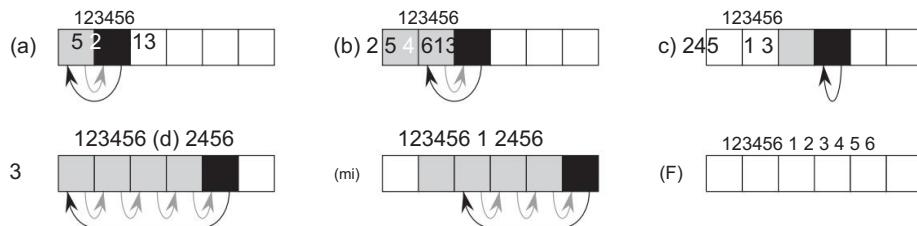


Figura 2.2 La operación de INSERCIÓN-CLASIFICACIÓN en el arreglo AD h5; 2; 4; 6; 1; 3i. Los índices de la matriz aparecen encima de los rectángulos y los valores almacenados en las posiciones de la matriz aparecen dentro de los rectángulos. (a)–(e) Las iteraciones del ciclo for de las líneas 1–8. En cada iteración, el rectángulo negro contiene la clave tomada de  $A[i]$ , que se compara con los valores de los rectángulos sombreados a su izquierda en la prueba de la línea 5. Las flechas sombreadas muestran los valores de matriz movidos una posición a la derecha en la línea 6, y los valores negros las flechas indican dónde se mueve la clave en la línea 8. (f) La matriz ordenada final.

### INSERTION-SORT.A/ 1

```

for j D 2 to A:length key D A[i]
    2 // Inserta
    A[i] en la secuencia ordenada A[1 : : i - 1]. i D j - 1 while i > 0 and A[i] >
    4         key A[i]
    5         1 D A[i] i D i A[i] C 1 tecla D
    6
    7             1
    8

```

### Invariantes de bucle y la corrección del ordenamiento por inserción

La Figura 2.2 muestra cómo funciona este algoritmo para AD h5; 2; 4; 6; 1; 3i. El índice  $j$  indica la “carta actual” que se está insertando en la mano. Al comienzo de cada iteración del ciclo for , que está indexado por  $j$  , el subarreglo que consta de los elementos  $A[1 : : j - 1]$  constituye la mano clasificada actualmente, y el subarreglo restante  $A[j : : n]$  corresponde a la pila de cartas que aún está en el mesa. De hecho, los elementos  $A[1 : : j - 1]$  son los elementos originalmente en las posiciones 1 a  $j - 1$ , pero ahora en orden ordenado. Enunciamos estas propiedades de  $A[1 : : j - 1]$  formalmente como un bucle invariante:

Al comienzo de cada iteración del bucle for de las líneas 1 a 8, el subarreglo  $A[1 : : j - 1]$  consta de los elementos que originalmente estaban en  $A[1 : : j - 1]$ , pero ordenados.

Usamos bucles invariantes para ayudarnos a comprender por qué un algoritmo es correcto. Debemos mostrar tres cosas sobre un bucle invariante:

Inicialización: es cierto antes de la primera iteración del ciclo.

Mantenimiento: si es verdadero antes de una iteración del ciclo, permanece verdadero antes de la próxima iteración.

Terminación: cuando el ciclo termina, el invariante nos da una propiedad útil  
eso ayuda a mostrar que el algoritmo es correcto.

Cuando se cumplen las dos primeras propiedades, la invariante del bucle es verdadera antes de cada iteración del bucle. (Por supuesto, somos libres de usar hechos establecidos que no sean la invariante del ciclo en sí misma para demostrar que la invariante del ciclo sigue siendo verdadera antes de cada iteración).

Tenga en cuenta la similitud con la inducción matemática, donde para probar que se cumple una propiedad, prueba un caso base y un paso inductivo. Aquí, mostrar que el invariante se cumple antes de la primera iteración corresponde al caso base, y mostrar que el invariante se cumple de iteración en iteración corresponde al paso inductivo.

La tercera propiedad es quizás la más importante, ya que estamos usando el bucle invariante para mostrar la corrección. Por lo general, usamos la invariante del bucle junto con la condición que provocó la finalización del bucle. La propiedad de terminación difiere de cómo solemos usar la inducción matemática, en la que aplicamos el paso inductivo infinitamente; aquí, detenemos la "inducción" cuando termina el bucle.

Veamos cómo se mantienen estas propiedades para el ordenamiento por inserción.

Inicialización: comenzamos mostrando que el ciclo invariante se mantiene antes de la primera iteración del ciclo, cuando  $j \leq 2$ .<sup>1</sup>

El subarreglo  $A[1:j]$  consta del único elemento  $A[1]$ , que de hecho es el elemento original en  $A[1]$ .

Además, este subarreglo está ordenado (trivialmente, por supuesto), lo que muestra que el ciclo invariante se mantiene antes de la primera iteración del ciclo.

Mantenimiento: A continuación, abordamos la segunda propiedad: mostrar que cada iteración mantiene el bucle invariante. De manera informal, el cuerpo del bucle `for` funciona moviendo  $A[j]$ ,  $A[j+1]$ ,  $A[j+2]$ , y así sucesivamente una posición a la derecha hasta que encuentra la posición adecuada para  $A[j]$  (líneas 4 a 7), momento en el cual inserta el valor de  $A[j]$  (línea 8). El subarreglo  $A[1:j]$  consta entonces de los elementos originalmente en  $A[1:j]$ , pero en orden ordenado. Incrementar  $j$  para la siguiente iteración del bucle `for` conserva el bucle invariante.

Un tratamiento más formal de la segunda propiedad requeriría que establezcamos y mostremos un ciclo invariante para el ciclo `while` de las líneas 5–7. En este punto, sin embargo,

<sup>1</sup>Cuando el ciclo es un ciclo `for`, el momento en el que verificamos el invariante del ciclo justo antes de la primera iteración es inmediatamente después de la asignación inicial a la variable del contador del ciclo y justo antes de la primera prueba en el encabezado del ciclo. En el caso de `INSERTION-SORT`, este tiempo es después de asignar 2 a la variable  $j$  pero antes de la primera prueba de si  $j < A.length$ .

preferimos no atascarnos en tal formalismo, por lo que nos basamos en nuestro análisis informal para mostrar que la segunda propiedad se cumple para el ciclo externo.

Terminación: finalmente, examinamos lo que sucede cuando termina el ciclo. La condición que hace que el ciclo for termine es que  $j > A:\text{longitud } D\ n$ . Debido a que cada iteración del bucle aumenta  $j$  en 1, debemos tener  $j \leq n$  en ese momento.

Sustituyendo  $n$  por  $j$  en la redacción del bucle invariante, tenemos que el subarreglo  $A[1 : j]$  consta de los elementos originalmente en  $A[1 : n]$ , pero en orden ordenado. Al observar que el subarreglo  $A[1 : n]$  es el arreglo completo, concluimos que el arreglo completo está ordenado. Por lo tanto, el algoritmo es correcto.

Usaremos este método de invariantes de bucle para mostrar la corrección más adelante en este capítulo y en otros capítulos también.

#### Convenciones de pseudocódigo

Usamos las siguientes convenciones en nuestro pseudocódigo.

La sangría indica la estructura del bloque. Por ejemplo, el cuerpo del ciclo for que comienza en la línea 1 consta de las líneas 2 a la 8, y el cuerpo del ciclo while que comienza en la línea 5 contiene las líneas 6 a 7 pero no la línea 8. Nuestro estilo de sangría se aplica a if- else declaraciones<sup>2</sup> también. El uso de la sangría en lugar de los indicadores convencionales de la estructura del bloque, como las declaraciones de inicio y finalización , reduce en gran medida el desorden al tiempo que preserva, o incluso mejora ,

la claridad . similares a los de C, C++, Java, Python y Pascal.<sup>4</sup> En este libro, el contador de bucle conserva su valor después de salir del bucle, a diferencia de algunas situaciones que surgen en C++, Java y Pascal. Por lo tanto, inmediatamente después de un ciclo for , el valor del contador del ciclo es el valor que primero excedió el límite del ciclo for . Usamos esta propiedad en nuestro argumento de corrección para el ordenamiento por inserción. El encabezado del ciclo for en la línea 1 es para  $j \leq n$  ( $j < n$  en A:longitud), por lo que cuando este ciclo termina,  $j = n$  (o, de manera equivalente,  $j = n + 1$ , ya que  $n = A:\text{longitud}$ ). Usamos la palabra clave to cuando un ciclo for incrementa su ciclo

<sup>2</sup>En una declaración if-else , sangramos else al mismo nivel que su if coincidente. Aunque omitimos la palabra clave entonces, ocasionalmente nos referimos a la parte ejecutada cuando la prueba que sigue a if es verdadera como una cláusula entonces. Para las pruebas de varias vías, usamos elseif para las pruebas posteriores a la primera.

<sup>3</sup>Cada procedimiento de pseudocódigo en este libro aparece en una página para que no tenga que discernir los niveles de sangría en el código que se divide en páginas.

<sup>4</sup>La mayoría de los lenguajes estructurados en bloques tienen construcciones equivalentes, aunque la sintaxis exacta puede diferir. Python carece de bucles de repetición hasta que funciona , y sus bucles for funcionan de forma un poco diferente a los bucles for de este libro.

contador en cada iteración, y usamos la palabra clave `downto` cuando un bucle `for` disminuye su contador de bucle. Cuando el contador de bucle cambia en una cantidad superior a 1, la cantidad de cambio sigue a la palabra clave opcional `by`.

El símbolo `"/"` indica que el resto de la línea es un comentario.

Una asignación múltiple de la forma `i D j D e` asigna a ambas variables `i` y `j` el valor de la expresión `e`; debe tratarse como equivalente a la asignación `j D e` seguida de la asignación `i D j`.

Las variables (como `i`, `j` y `key`) son locales para el procedimiento dado. No utilizaremos variables globales sin indicación explícita.

Accedemos a los elementos de la matriz especificando el nombre de la matriz seguido del índice entre corchetes. Por ejemplo, `A[i]` indica el  $i$ -ésimo elemento del arreglo `A`. La notación `:` se usa para indicar un rango de valores dentro de un arreglo. Así, `A[1 : : j]` indica el subarreglo de `A` que consta de los  $j$  elementos `A[1]; A[2]; \dots; A[j]`.

Por lo general, organizamos datos compuestos en objetos, que se componen de atributos.

Accedemos a un atributo en particular utilizando la sintaxis que se encuentra en muchos lenguajes de programación orientados a objetos: el nombre del objeto, seguido de un punto, seguido del nombre del atributo. Por ejemplo, tratamos una matriz como un objeto con el atributo de longitud que indica cuántos elementos contiene. Para especificar el número de elementos en una matriz `A`, escribimos `A:length`.

Tratamos una variable que representa una matriz u objeto como un puntero a los datos que representan la matriz u objeto. Para todos los atributos `f` de un objeto `x`, establecer `y D x` hace que `y:f` sea igual a `x:f`. Además, si ahora establecemos `x:f D 3`, luego no solo `x:f` es igual a 3, sino que `y:f` es igual a 3 también. En otras palabras, `x` e `y` apuntan al mismo objeto después de la asignación `y D x`.

Nuestra notación de atributos puede "en cascada". Por ejemplo, suponga que el atributo `f` es en sí mismo un puntero a algún tipo de objeto que tiene un atributo `g`. Entonces la notación `x:f:g` está implícitamente entre paréntesis como `.x:f:g`. En otras palabras, si hubiésemos asignado `y D x:f`, entonces `x:f:g` es lo mismo que `y:g`.

A veces, un puntero no se referirá a ningún objeto en absoluto. En este caso, le damos el valor especial `NIL`.

Pasamos parámetros a un procedimiento por valor: el procedimiento llamado recibe su propia copia de los parámetros, y si asigna un valor a un parámetro, el procedimiento que llama no ve el cambio. Cuando se pasan objetos, se copia el puntero a los datos que representan el objeto, pero no los atributos del objeto. Por ejemplo, si `x` es un parámetro de un procedimiento llamado, la asignación `x D y` dentro del procedimiento llamado no es visible para el procedimiento que llama. La asignación `x:f D 3`, sin embargo, es visible. De manera similar, las matrices se pasan por puntero, de modo que

se pasa un puntero a la matriz, en lugar de la matriz completa, y los cambios en los elementos individuales de la matriz son visibles para el procedimiento de llamada.

Una declaración de devolución transfiere inmediatamente el control al punto de llamada en el procedimiento de llamada. La mayoría de las declaraciones de devolución también toman un valor para devolverlo a la persona que llama. Nuestro pseudocódigo se diferencia de muchos lenguajes de programación en que permitimos que se devuelvan múltiples valores en una sola declaración de devolución .

Los operadores booleanos "y" y "o" están en cortocircuito. Es decir, cuando evaluamos la expresión "x e y" primero evaluamos x. Si x se evalúa como FALSO, entonces la expresión completa no se puede evaluar como VERDADERO, por lo que no evaluamos y.

Si, por otro lado, x se evalúa como VERDADERO, debemos evaluar y para determinar el valor de la expresión completa. De manera similar, en la expresión "x o y" evaluamos la expresión y solo si x se evalúa como FALSO. Los operadores de cortocircuito nos permiten escribir expresiones booleanas como "x ≠ NIL y x:f D y" sin preocuparnos de lo que sucede cuando tratamos de evaluar x:f cuando x es NIL..

La palabra clave error indica que se produjo un error porque las condiciones eran incorrectas para que se llamara al procedimiento. El procedimiento de llamada es responsable de manejar el error, por lo que no especificamos qué acción tomar.

## Ejercicios

### 2.1-1

Usando la Figura 2.2 como modelo, ilustre la operación de INSERCIÓN-CLASIFICACIÓN en el arreglo AD h31; 41; 59; 26; 41; 58i.

### 2.1-2

Vuelva a escribir el procedimiento INSERCIÓN-CLASIFICACIÓN para clasificar en orden no creciente en lugar de no decreciente.

### 2.1-3

Considere el problema de búsqueda:

Entrada: Una secuencia de n números AD ha1; a2;:::;ani y un valor .

Salida: Un índice i tal que D AŒi o el valor especial NIL si no aparecen en a.

Escriba un pseudocódigo para la búsqueda lineal, que explora la secuencia en busca de . Usando un bucle invariante, demuestre que su algoritmo es correcto. Asegúrese de que su bucle invariante cumpla con las tres propiedades necesarias.

### 2.1-4

Considere el problema de sumar dos enteros binarios de n bits, almacenados en dos matrices A y B de n elementos. La suma de los dos enteros debe almacenarse en forma binaria en

una matriz de  $n \times n$  C 1-elemento C. Plantee el problema formalmente y escriba un pseudocódigo para sumar los dos enteros.

---

## 2.2 Analizando algoritmos

Analizar un algoritmo ha llegado a significar predecir los recursos que requiere el algoritmo.

Ocasionalmente, los recursos como la memoria, el ancho de banda de comunicación o el hardware de la computadora son la principal preocupación, pero con mayor frecuencia es el tiempo de cómputo lo que queremos medir. Generalmente, al analizar varios algoritmos candidatos para un problema, podemos identificar el más eficiente. Dicho análisis puede indicar más de un candidato viable, pero a menudo podemos descartar varios algoritmos inferiores en el proceso.

Antes de que podamos analizar un algoritmo, debemos tener un modelo de la tecnología de implementación que usaremos, incluido un modelo para los recursos de esa tecnología y sus costos. Durante la mayor parte de este libro, supondremos un modelo genérico de computación de máquina de acceso aleatorio (RAM) de un procesador como nuestra tecnología de implementación y comprenderemos que nuestros algoritmos se implementarán como programas de computadora. En el modelo RAM, las instrucciones se ejecutan una tras otra, sin operaciones simultáneas.

En rigor, deberíamos definir con precisión las instrucciones del modelo RAM y sus costos. Sin embargo, hacerlo sería tedioso y arrojaría poca información sobre el diseño y análisis de algoritmos. Sin embargo, debemos tener cuidado de no abusar del modelo RAM. Por ejemplo, ¿qué pasaría si una RAM tuviera una instrucción que ordenara? Entonces podríamos ordenar en una sola instrucción. Tal RAM sería poco realista, ya que las computadoras reales no tienen tales instrucciones. Nuestra guía, por lo tanto, es cómo se diseñan las computadoras reales. El modelo RAM contiene instrucciones que se encuentran comúnmente en computadoras reales: aritmética (como sumar, restar, multiplicar, dividir, resto, piso, techo), movimiento de datos (cargar, almacenar, copiar) y control (rama condicional e incondicional, llamada a subrutina). y volver). Cada instrucción requiere una cantidad constante de tiempo.

Los tipos de datos en el modelo RAM son enteros y coma flotante (para almacenar números reales). Aunque normalmente no nos preocupamos por la precisión en este libro, en algunas aplicaciones la precisión es crucial. También asumimos un límite en el tamaño de cada palabra de datos. Por ejemplo, cuando trabajamos con entradas de tamaño  $n$ , por lo general asumimos que los números enteros están representados por  $c \lg n$  bits para alguna constante  $c > 1$ .

Requerimos  $c > 1$  para que cada palabra pueda contener el valor de  $n$ , lo que nos permite indexar los elementos de entrada individuales, y restringimos  $c$  para que sea una constante para que el tamaño de la palabra no crezca arbitrariamente. (Si el tamaño de la palabra pudiera crecer arbitrariamente, podríamos almacenar grandes cantidades de datos en una palabra y operar con todo en tiempo constante, claramente un escenario poco realista).

Las computadoras reales contienen instrucciones que no figuran en la lista anterior y tales instrucciones representan un área gris en el modelo de RAM. Por ejemplo, ¿la exponenciación es una instrucción de tiempo constante? En el caso general, no; se necesitan varias instrucciones para calcular  $x^y$  cuando  $x$  e  $y$  son números reales. Sin embargo, en situaciones restringidas, la exponenciación es una operación de tiempo constante. Muchas computadoras tienen una instrucción de "desplazamiento a la izquierda", que en tiempo constante desplaza los bits de un número entero  $k$  posiciones hacia la izquierda. En la mayoría de las computadoras, desplazar los bits de un entero una posición a la izquierda equivale a multiplicar por 2, de modo que desplazar los bits  $k$  posiciones a la izquierda equivale a multiplicar por  $2^k$ . Por lo tanto, tales computadoras pueden calcular  $2^k$  en una instrucción de tiempo constante desplazando el número entero 1  $k$  posiciones hacia la izquierda, siempre que  $k$  no sea más que el número de bits en una palabra de computadora. Nos esforzaremos por evitar tales áreas grises en el modelo RAM, pero trataremos el cálculo de  $2^k$  como una operación de tiempo constante cuando  $k$  es un entero positivo lo suficientemente pequeño.

En el modelo RAM, no intentamos modelar la jerarquía de memoria que es común en las computadoras contemporáneas. Es decir, no modelamos cachés ni memoria virtual. Varios modelos computacionales intentan dar cuenta de los efectos de la jerarquía de memoria, que a veces son significativos en programas reales en máquinas reales. Un puñado de problemas en este libro examinan los efectos de la jerarquía de la memoria, pero en su mayor parte, los análisis de este libro no los considerarán. Los modelos que incluyen la jerarquía de memoria son un poco más complejos que el modelo de RAM, por lo que puede ser difícil trabajar con ellos. Además, los análisis del modelo RAM suelen ser excelentes predictores del rendimiento en máquinas reales.

Analizar incluso un algoritmo simple en el modelo RAM puede ser un desafío. Las herramientas matemáticas requeridas pueden incluir combinatoria, teoría de la probabilidad, destreza algebraica y la capacidad de identificar los términos más significativos en una fórmula.

Debido a que el comportamiento de un algoritmo puede ser diferente para cada entrada posible, necesitamos un medio para resumir ese comportamiento en fórmulas simples y fáciles de entender.

Aunque normalmente seleccionamos solo un modelo de máquina para analizar un algoritmo dado, aún enfrentamos muchas opciones al decidir cómo expresar nuestro análisis. Nos gustaría una forma que sea fácil de escribir y manipular, que muestre las características importantes de los requisitos de recursos de un algoritmo y suprima los detalles tediosos.

#### Análisis de ordenación por inserción

El tiempo que tarda el procedimiento INSERCIÓN-CLASIFICACIÓN depende de la entrada: clasificar mil números lleva más tiempo que clasificar tres números. Además, INSERTION SORT puede tomar diferentes cantidades de tiempo para ordenar dos secuencias de entrada del mismo tamaño dependiendo de qué tan cerca estén ya ordenadas. En general, el tiempo que tarda un algoritmo crece con el tamaño de la entrada, por lo que es tradicional describir el tiempo de ejecución de un programa en función del tamaño de su entrada. Para hacerlo, debemos definir los términos "tiempo de ejecución" y "tamaño de entrada" con más cuidado.

La mejor noción para el tamaño de entrada depende del problema que se esté estudiando. Para muchos problemas, como ordenar o calcular transformadas discretas de Fourier, la medida más natural es el número de elementos en la entrada, por ejemplo, el tamaño de matriz  $n$  para ordenar. Para muchos otros problemas, como multiplicar dos números enteros, la mejor medida del tamaño de entrada es el número total de bits necesarios para representar la entrada en notación binaria ordinaria. A veces, es más apropiado describir el tamaño de la entrada con dos números en lugar de uno. Por ejemplo, si la entrada de un algoritmo es un gráfico, el tamaño de la entrada se puede describir mediante el número de vértices y aristas del gráfico. Indicaremos qué medida de tamaño de entrada se está utilizando con cada problema que estudiemos.

El tiempo de ejecución de un algoritmo en una entrada particular es el número de operaciones primitivas o "pasos" ejecutados. Es conveniente definir la noción de paso de modo que sea lo más independiente posible de la máquina. Por el momento, adoptemos el siguiente punto de vista. Se requiere una cantidad de tiempo constante para ejecutar cada línea de nuestro pseudocódigo. Una línea puede tomar una cantidad de tiempo diferente a otra línea, pero supondremos que cada ejecución de la  $i$ -ésima línea toma un tiempo  $c_i$ , donde  $c_i$  es una constante. Este punto de vista está de acuerdo con el modelo RAM y también refleja cómo se implementaría el pseudocódigo en la mayoría de las computadoras reales.<sup>5</sup>

En la siguiente discusión, nuestra expresión para el tiempo de ejecución de INSERTION-SORT evolucionará de una fórmula desordenada que usa todos los costos de declaración  $c_i$  a una notación mucho más simple que es más concisa y más fácil de manipular. Esta notación más simple también facilitará determinar si un algoritmo es más eficiente que otro.

Comenzamos presentando el procedimiento INSERTION-SORT con el "costo" de tiempo de cada sentencia y el número de veces que se ejecuta cada sentencia. Para cada  $j \ D\ 2; 3; : : : ; n$ , donde  $n \ D\ A.length$ , dejamos que  $t_j$  denote el número de veces que se ejecuta la prueba de ciclo while en la línea 5 para ese valor de  $j$ . Cuando un bucle for o while sale de la forma habitual (es decir, debido a la prueba en el encabezado del bucle), la prueba se ejecuta una vez más que el cuerpo del bucle. Suponemos que los comentarios no son sentencias ejecutables, por lo que no toman tiempo.

<sup>5</sup>Hay algunas sutilezas aquí. Los pasos computacionales que especificamos en inglés son a menudo variantes de un procedimiento que requiere algo más que una cantidad constante de tiempo. Por ejemplo, más adelante en este libro podríamos decir "ordenar los puntos por la coordenada  $x$ ", lo que, como veremos, lleva más de una cantidad constante de tiempo. Además, tenga en cuenta que una sentencia que llama a una subrutina lleva un tiempo constante, aunque la subrutina, una vez invocada, puede tardar más. Es decir, separamos el proceso de llamar a la subrutina (pasarle parámetros, etc.) del proceso de ejecutar la subrutina.

		costo	veces
INSERTION-SORT.A/ 1			
for j D 2 to A:length 2 key D		c1	<small>norte</small>
AŒj 3 // Inserta		c2	n 1
AŒj en la secuencia ordenada AŒ1 : : j			
1. i D j 1 while i>0 and		0	n 1 n
4. AŒi > key		c4	1 c5
5. AŒi C 1 D AŒi		Pn jD2 tj c6Pn	
6.		jD2.tj 1/ c7 Pn jD2.tj 1/	
7.       yo re yo     1			
8. Tecla AŒi C 1 D		c8	n 1

El tiempo de ejecución del algoritmo es la suma de los tiempos de ejecución de cada sentencia ejecutada; una instrucción que toma  $c_i$  pasos para ejecutarse y ejecuta  $n$  veces contribuirá con  $c_i n$  al tiempo total de ejecución.<sup>6</sup> Para calcular  $T .n/$ , el tiempo de ejecución de INSERTION-SORT en una entrada de  $n$  valores, sumamos los productos del costo y columnas de tiempos , obteniendo

$$T .n/ = D c1n C c2.n 1/ C c4.n 1/ C c5 Xn tj C c6 Xn C c7 Xn \cdot tj .1/ \\ jD2 jD2$$

$$\cdot tj .1/ C c8.n 1/ : \\ jD2$$

Incluso para entradas de un tamaño dado, el tiempo de ejecución de un algoritmo puede depender de qué entrada de ese tamaño se proporcione. Por ejemplo, en INSERTION-SORT, el mejor caso ocurre si la matriz ya está ordenada. Para cada  $j D 2; 3; : : : ; n$ , luego encontramos que  $AŒi$  clave en la línea 5 cuando  $i$  tiene su valor inicial de  $j 1$ . Así  $tj D 1$  para  $j D 2; 3; : : : ; n$ , y el tiempo de ejecución en el mejor de los casos es

$$T .n/ = D c1n C c2.n 1/ C c4.n 1/ C c5.n 1/ C c8.n 1/ D .c1 C c2 C c4 C c5 C \\ c8/n .c2 C c4 C c5 C c8/ :$$

Podemos expresar este tiempo de ejecución como un  $C b$  para las constantes ayb que dependen de los costos de declaración  $c_i$ ; por tanto, es una función lineal de  $n$ .

Si la matriz está en orden inverso, es decir, en orden decreciente, se obtiene el peor de los casos. Debemos comparar cada elemento  $AŒj$  con cada elemento en todo el subarreglo ordenado  $AŒ1 : : j 1$ , y así  $tj D j$  para  $j D 2; 3; : : : ; n$ . Señalando que

<sup>6</sup>Esta característica no se cumple necesariamente para un recurso como la memoria. Una declaración que hace referencia a  $m$  palabras de memoria y se ejecuta  $n$  veces no necesariamente hace referencia a  $mn$  palabras de memoria distintas.

$$\frac{x_n}{jD_2} \cdot D_2 \cdot \frac{nn C 1/j}{2} = 1$$

y

$$\frac{x_n}{jD_2} \cdot \frac{1/D}{2} \cdot \frac{nn 1/j}{2}$$

(vea el Apéndice A para una revisión de cómo resolver estas sumas), encontramos que en el peor de los casos, el tiempo de ejecución de INSERTION-SORT es

$$T \cdot n / D c_1 n C c_2 n 1 / C c_4 n 1 / C c_5 = \frac{nn C 1/2}{2} + 1$$

$$= do c_6 \cdot \frac{nn 1/2}{2} + do c_7 \cdot \frac{nn 1/2}{2} + C c_8 n 1 /$$

$$= \frac{c_5 c_6 c_7}{2^2 2^2 \cdot c_2} C - n^2 C c_1 C c_2 C c_4 C + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} C c_8 \text{ norte}$$

$$+ c_4 C c_5 C c_8 :$$

Podemos expresar este tiempo de ejecución en el peor de los casos como  $a n^2 C b n C c$  para las constantes  $a$ ,  $b$  y  $c$  que nuevamente dependen de los costos de declaración  $c_i$ ; por tanto, es una función cuadrática de  $n$ .

Típicamente, como en el ordenamiento por inserción, el tiempo de ejecución de un algoritmo es fijo para una entrada dada, aunque en capítulos posteriores veremos algunos algoritmos "aleatorizados" interesantes cuyo comportamiento puede variar incluso para una entrada fija.

### Análisis de el peor caso y caso promedio

En nuestro análisis de la ordenación por inserción, analizamos tanto el mejor caso, en el que la matriz de entrada ya estaba ordenada, como el peor de los casos, en el que la matriz de entrada estaba ordenada inversamente. Sin embargo, en el resto de este libro, por lo general, nos concentraremos en encontrar solo el tiempo de ejecución del peor de los casos, es decir, el tiempo de ejecución más largo para cualquier entrada de tamaño  $n$ . Damos tres razones para esta orientación.

El tiempo de ejecución del peor de los casos de un algoritmo nos da un límite superior en el tiempo de ejecución para cualquier entrada. Saberlo proporciona una garantía de que el algoritmo nunca tardará más. No necesitamos hacer una conjetura informada sobre el tiempo de ejecución y esperar que nunca empeore mucho.

Para algunos algoritmos, el peor de los casos ocurre con bastante frecuencia. Por ejemplo, al buscar en una base de datos una determinada información, el peor de los casos del algoritmo de búsqueda suele ocurrir cuando la información no está presente en la base de datos. En algunas aplicaciones, las búsquedas de información ausente pueden ser frecuentes.

El "caso promedio" suele ser tan malo como el peor de los casos. Supongamos que elegimos aleatoriamente  $n$  números y aplicamos ordenación por inserción. ¿Cuánto tiempo lleva determinar en qué parte del subarreglo  $A[1:n]$  insertar el elemento  $A[j]$ ? En promedio, la mitad de los elementos en  $A[1:n]$  son menores que  $A[j]$ , y la mitad de los elementos son mayores. En promedio, por lo tanto, comprobamos la mitad del subarreglo  $A[1:n]$ , por lo que  $t_j$  es aproximadamente  $j=2$ . El tiempo de ejecución del caso promedio resultante resulta ser una función cuadrática del tamaño de entrada, al igual que el tiempo de ejecución del peor de los casos.

En algunos casos particulares, nos interesará el tiempo de ejecución de caso promedio de un algoritmo; Veremos la técnica del análisis probabilístico aplicada a varios algoritmos a lo largo de este libro. El alcance del análisis de casos promedio es limitado, porque puede no ser evidente lo que constituye una entrada "promedio" para un problema en particular. A menudo, supondremos que todas las entradas de un tamaño dado son igualmente probables. En la práctica, esta suposición puede violarse, pero a veces podemos usar un algoritmo aleatorio, que hace elecciones aleatorias, para permitir un análisis probabilístico y producir un tiempo de ejecución esperado . Exploramos más los algoritmos aleatorios en el Capítulo 5 y en varios otros capítulos posteriores.

#### orden de crecimiento

Usamos algunas abstracciones simplificadoras para facilitar nuestro análisis del procedimiento CLASIFICACIÓN POR INSERCIÓN . Primero, ignoramos el costo real de cada declaración, usando las constantes  $c_i$  para representar estos costos. Luego, observamos que incluso estas constantes nos brindan más detalles de los que realmente necesitamos: expresamos el tiempo de ejecución en el peor de los casos como  $a n^2 + b n + c$  para algunas constantes  $a, b$  y  $c$  que dependen del enunciado cuesta  $c_i$ . Por lo tanto, ignoramos no solo los costos de declaración reales, sino también los costos abstractos  $c_i$ .

Ahora haremos una abstracción simplificadora más: es la tasa de crecimiento, o el orden de crecimiento, del tiempo de ejecución lo que realmente nos interesa. Por lo tanto, consideramos sólo el término principal de una fórmula (p. ej.,  $a n^2$ ), ya que los términos de orden inferior son relativamente insignificantes para valores grandes de  $n$ . También ignoramos el coeficiente constante del término principal, ya que los factores constantes son menos significativos que la tasa de crecimiento para determinar la eficiencia computacional para entradas grandes. Para el ordenamiento por inserción, cuando ignoramos los términos de orden inferior y el coeficiente constante del término principal, nos quedamos con el factor de  $n^2$  del término principal. Escribimos que la ordenación por inserción tiene un tiempo de ejecución en el peor de los casos de  $\Theta(n^2)$  (pronunciado "theta of  $n$ -squared"). Usaremos la notación  $\Theta$ , de manera informal en este capítulo y la definiremos con precisión en el Capítulo 3.

Por lo general, consideramos que un algoritmo es más eficiente que otro si su tiempo de ejecución en el peor de los casos tiene un orden de crecimiento más bajo. Debido a los factores constantes y a los términos de orden inferior, un algoritmo cuyo tiempo de ejecución tiene un orden de crecimiento más alto puede requerir menos tiempo para entradas pequeñas que un algoritmo cuyo tiempo de ejecución tiene un orden de crecimiento más bajo.

orden de crecimiento. Pero para entradas lo suficientemente grandes, un algoritmo „n<sup>2</sup>“, por ejemplo, se ejecutará más rápido en el peor de los casos que un algoritmo „n<sup>3</sup>“.

### Ejercicios

#### 2.2-1

Exprese la función  $n^3 = 1000 \cdot 100n^2 \cdot 100n \cdot C \cdot 3$  en términos de notación „„.

#### 2.2-2

Considere clasificar n números almacenados en el arreglo A encontrando primero el elemento más pequeño de A e intercambiándolo con el elemento en A[0]. Luego encuentre el segundo elemento más pequeño de A e intercambíelo con A[1]. Continúe de esta manera para los primeros n<sub>1</sub> elementos de A. Escriba un pseudocódigo para este algoritmo, que se conoce como clasificación por selección. ¿Qué bucle invariante mantiene este algoritmo? ¿Por qué necesita ejecutarse solo para los primeros n<sub>1</sub> elementos, en lugar de para todos los n elementos? Proporcione los tiempos de ejecución del mejor y el peor caso del tipo de selección en notación „„.

#### 2.2-3

Considere nuevamente la búsqueda lineal (vea el Ejercicio 2.1-3). ¿Cuántos elementos de la secuencia de entrada deben verificarse en promedio, suponiendo que el elemento que se busca tiene la misma probabilidad de ser cualquier elemento de la matriz? ¿Qué tal en el peor de los casos? ¿Cuáles son los tiempos de ejecución del caso promedio y del caso más desfavorable de la búsqueda lineal en notación „„? Justifique sus respuestas.

#### 2.2-4

¿Cómo podemos modificar casi cualquier algoritmo para tener un buen tiempo de ejecución en el mejor de los casos?

## 2.3 Diseño de algoritmos

Podemos elegir entre una amplia gama de técnicas de diseño de algoritmos. Para la ordenación por inserción, usamos un enfoque incremental : después de ordenar el subarreglo A[1 : : j], insertamos el elemento único A[j] en su lugar adecuado, lo que produce el subarreglo ordenado A[1 : : j].

En esta sección, examinamos un enfoque de diseño alternativo, conocido como "divide y vencerás", que exploraremos con más detalle en el Capítulo 4. Usaremos divide y vencerás para diseñar un algoritmo de clasificación cuyo tiempo de ejecución en el peor de los casos es mucho menor que el del tipo de inserción. Una ventaja de los algoritmos de divide y vencerás es que sus tiempos de ejecución a menudo se determinan fácilmente usando técnicas que veremos en el Capítulo 4.

### 2.3.1 El enfoque divide y vencerás

Muchos algoritmos útiles tienen una estructura recursiva : para resolver un problema dado, se llaman a sí mismos recursivamente una o más veces para tratar subproblemas estrechamente relacionados. Estos algoritmos generalmente siguen un enfoque de divide y vencerás : dividen el problema en varios subproblemas que son similares al problema original pero de menor tamaño, resuelven los subproblemas recursivamente y luego combinan estas soluciones para crear una solución al problema original. .

El paradigma divide y vencerás involucra tres pasos en cada nivel de la recursión:

Dividir el problema en varios subproblemas que son instancias más pequeñas del mismo problema.

Conquistar los subproblemas resolviéndolos recursivamente. Sin embargo, si los tamaños de los subproblemas son lo suficientemente pequeños, simplemente resuelva los subproblemas de una manera directa.

Combinar las soluciones de los subproblemas en la solución del problema original  
lema

El algoritmo de clasificación por fusión sigue de cerca el paradigma divide y vencerás. Intuitivamente, funciona de la siguiente manera.

Dividir: divide la secuencia de n elementos que se va a clasificar en dos subsecuencias de n=2 elementos cada uno.

Conquer: ordena las dos subsecuencias recursivamente usando la ordenación por combinación.

Combine: combine las dos subsecuencias ordenadas para producir la respuesta ordenada.

La recursividad "toca fondo" cuando la secuencia que se va a ordenar tiene una longitud de 1, en cuyo caso no hay trabajo que hacer, ya que cada secuencia de longitud 1 ya está ordenada.

La operación clave del algoritmo de clasificación por fusión es la fusión de dos secuencias ordenadas en el paso de "combinar". Fusionamos llamando a un procedimiento auxiliar MERGE(A; pag; q; r), donde A es una matriz y p, q y r son índices de la matriz tales que p < q < r. El procedimiento asume que los subarreglos A[ $p:p-1$ ] y A[ $q:q-1$ ] están ordenados. Los fusiona para formar un solo subarreglo ordenado que reemplaza el subarreglo actual A[ $p:r$ ]:

R.

Nuestro procedimiento MERGE toma tiempo  $n \log n$ , donde  $n$  es el número total de elementos que se fusionan, y funciona de la siguiente manera. Volviendo a nuestro motivo de juego de cartas, supongamos que tenemos dos pilas de cartas boca arriba sobre una mesa. Cada pila está ordenada, con las cartas más pequeñas en la parte superior. Deseamos fusionar las dos pilas en una sola pila de salida clasificada, que debe estar boca abajo sobre la mesa. Nuestro paso básico consiste en elegir la más pequeña de las dos cartas en la parte superior de las pilas boca arriba, quitarla de su pila (lo que expone una nueva carta superior) y colocar esta carta boca abajo en

la pila de salida. Repetimos este paso hasta que una pila de entrada esté vacía, momento en el que simplemente tomamos la pila de entrada restante y la colocamos boca abajo en la pila de salida.

Computacionalmente, cada paso básico lleva un tiempo constante, ya que estamos comparando solo las dos cartas superiores. Dado que realizamos como máximo  $n$  pasos básicos, la fusión lleva  $\sim n$  tiempo.

El siguiente pseudocódigo implementa la idea anterior, pero con un giro adicional que evita tener que verificar si alguna pila está vacía en cada paso básico.

Colocamos en la parte inferior de cada pila una tarjeta centinela, que contiene un valor especial que usamos para simplificar nuestro código. Aquí, usamos 1 como valor centinela, de modo que cada vez que se expone una carta con 1, no puede ser la carta más pequeña a menos que ambos montones tengan sus cartas centinela expuestas. Pero una vez que eso sucede, todas las cartas que no son centinela ya se han colocado en la pila de salida. Dado que sabemos de antemano que se colocarán exactamente  $r = C_1$  cartas en la pila de salida, podemos detenernos una vez que hayamos realizado tantos pasos básicos.

```
FUSIONAR(A; pag; q; r)
    1 n1 D qp C 1 2 n2 D rq
    3 sean LŒ1 :: n1
    C 1 y RŒ1 :: n2 C 1 nuevos arreglos 4 para i D 1 a n1

    LŒi D AŒp C i 5 6      1
    para j D 1 a n2
        RŒj D AŒq C j 7
        8 LŒn1 C 1 D 1
        9 RŒn2 C 1 D 1 10 i D 1

        11 j D 1 12
        para k D p to r 13 si LŒi
            RŒj
            14         AŒk D LŒi i
            15         D i C 1 else
            16         AŒk D RŒj j D j C 1
            17
```

En detalle, el procedimiento MERGE funciona de la siguiente manera. La línea 1 calcula la longitud  $n_1$  del subarreglo  $AŒp :: q$ , y la línea 2 calcula la longitud  $n_2$  del subarreglo  $AŒq C 1 :: r$ . Creamos matrices  $L$  y  $R$  ("izquierda" y "derecha"), de longitudes  $n_1 \leq C_1$  y  $n_2 \leq C_1$ , respectivamente, en la línea 3; la posición adicional en cada matriz mantendrá el centinela. El bucle for de las líneas 4–5 copia el subarreglo  $AŒp :: q$  en  $LŒi :: n_1$ , y el bucle for de las líneas 6–7 copia el subarreglo  $AŒq C 1 :: r$  en  $RŒi :: n_2$ .

Las líneas 8 y 9 colocan a los centinelas en los extremos de los arreglos  $L$  y  $R$ . Las líneas 10 a 17 ilustran

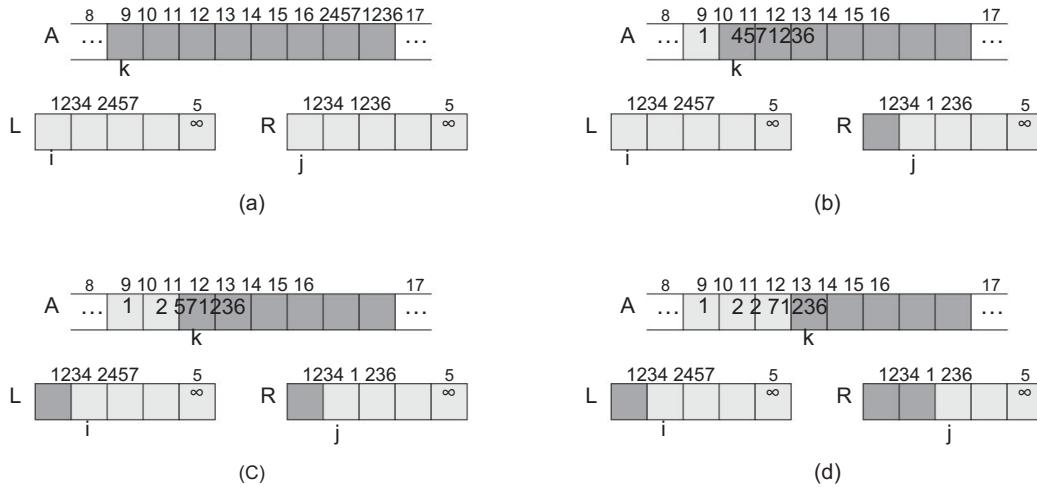


Figura 2.3 El funcionamiento de las líneas 10–17 en la llamada `MERGE.A; 9; 12; 16/`, cuando el subarreglo `A[9 : : 16]` contiene la secuencia `h2; 4; 5; 7; 1; 2; 3; 6`. Después de copiar e insertar centinelas, la matriz `L` contiene `h2; 4; 5; 7; 1`, y la matriz `R` contiene `h1; 2; 3; 6; 1`. Las posiciones ligeramente sombreadas en `A` contienen sus valores finales, y las posiciones ligeramente sombreadas en `L` y `R` contienen valores que aún no se han vuelto a copiar en `A`. En conjunto, las posiciones ligeramente sombreadas siempre comprenden los valores originales en `A[9 : : 16]`, junto con los dos centinelas. Las posiciones muy sombreadas en `A` contienen valores que se copiarán, y las posiciones muy sombreadas en `L` y `R` contienen valores que ya se han vuelto a copiar en `A`. (a)–(h) Las matrices `A`, `L` y `R`, y sus respectivos índices `k`, `i` y `j` antes de cada iteración del ciclo de las líneas 12–17.

Como se muestra en la figura 2.3, realice los pasos básicos de rp C1 manteniendo invariante el siguiente bucle:

Al comienzo de cada iteración del ciclo for de las líneas 12–17, el subarreglo `A[p : : k]` contiene los `k` elementos más pequeños de `L[i : : n1]` y `R[j : : n2]`, en orden ordenado. Además, `L[i : : k]` y `R[j : : k]` son los elementos más pequeños de sus arreglos que no han sido copiados nuevamente en `A`.

Debemos demostrar que esta invariante de bucle se mantiene antes de la primera iteración del bucle for de las líneas 12 a 17, que cada iteración del bucle mantiene la invariante y que la invariante proporciona una propiedad útil para mostrar la corrección cuando termina el bucle.

Inicialización: Antes de la primera iteración del bucle, tenemos `k = D(p)`, por lo que el subarreglo `A[p : : k]` está vacío. Este subarreglo vacío contiene los `k` elementos más pequeños de `L` y `R`, y dado que `i = D(1)`, tanto `L[i : : k]` como `R[j : : k]` son los elementos más pequeños de sus arreglos que no han sido copiados nuevamente en `A`.

## 2.3 Diseño de algoritmos

33

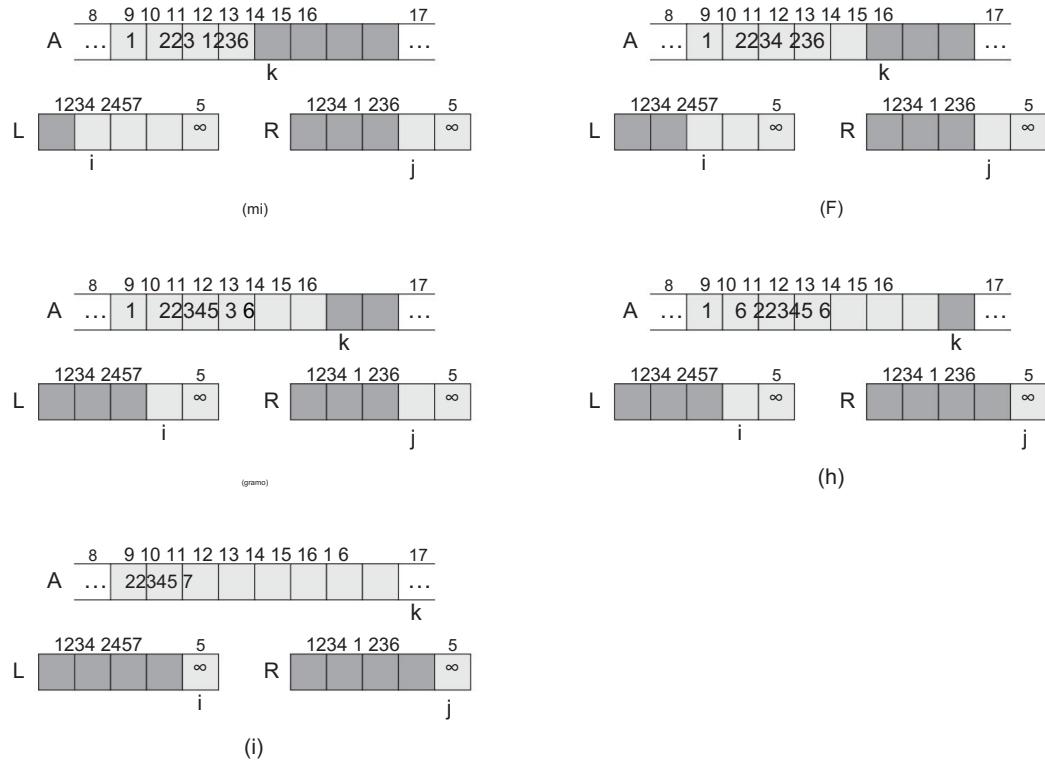


Figura 2.3, continuación (i) Las matrices y los índices en la terminación. En este punto, el subarreglo en  $A[9 : : 16]$  está ordenado, y los dos centinelas en L y R son los únicos dos elementos en estos arreglos que no han sido copiados en A.

**Mantenimiento:** Para ver que cada iteración mantiene el bucle invariante, supongamos primero que  $L[i] < R[j]$ . Entonces  $L[i]$  es el elemento más pequeño que aún no se ha vuelto a copiar en A. Como  $A[p : : k-1]$  contiene los  $k-p$  elementos más pequeños, después de que la línea 14 copie  $L[i]$  en  $A[k]$ , el subarreglo  $A[p : : k]$  contendrá los  $k-p$  1 elementos más pequeños. Incrementar  $k$  (en la actualización del bucle for) e  $i$  (en la línea 15) restablece el bucle invariable para la siguiente iteración. Si, en cambio,  $L[i] > R[j]$ , entonces las líneas 16 y 17 realizan la acción adecuada para mantener el bucle invariable.

**Terminación:** En la terminación,  $k \geq r+1$ . Por el bucle invariante, el subarreglo  $A[p : : k-1]$ , que es  $A[p : : r]$ , contiene los  $k-p$  elementos más pequeños de  $L[1 : : n1]$  y  $R[1 : : n2]$ , en orden ordenado. Los arreglos L y R juntos contienen  $n1+n2-2$  elementos. Todos menos los dos más grandes se han vuelto a copiar en A, y estos dos elementos más grandes son los centinelas.

Para ver que el procedimiento MERGE se ejecuta en tiempo  $,n/$ , donde  $n \geq C 1$ , observe que cada una de las líneas 1–3 y 8–11 toma un tiempo constante, los bucles for de las líneas 4–7 toman  $,n/ C n/2 D ,n/ \text{time}$ ,<sup>7</sup> y hay  $n$  iteraciones del ciclo for de las líneas 12–17, cada una de las cuales toma un tiempo constante.

Ahora podemos usar el procedimiento MERGE como una subrutina en el algoritmo de clasificación por fusión. El procedimiento MERGE-SORT.A; pag; r/ ordena los elementos en el subrayo AŒp :: r. Si pr, el subarreglo tiene como máximo un elemento y, por lo tanto, ya está ordenado. De lo contrario, el paso de dividir simplemente calcula un índice q que divide AŒp :: r en dos subarreglos: AŒp :: q, que contiene  $d_n=2e$  elementos, y AŒq C 1::r, que contiene  $b_n=2c$  elementos.<sup>8</sup>

MERGE-SORT.A; pag; r/ 1 si

```
p<r 2 q D pb
C r/=2c 3 MERGE-SORT.A; pag;
q/ 4 MERGE-SORT.A; q C 1; r/ 5
FUSIONAR.A; pag; q; r/
```

Para ordenar la secuencia completa AD hAŒ1; AŒ2; :: ; AŒni, hacemos la llamada inicial MERGE-SORT.A; 1; A:longitud/, donde una vez más A:longitud D n. La figura 2.4 ilustra el funcionamiento del procedimiento de abajo hacia arriba cuando n es una potencia de 2. El algoritmo consiste en fusionar pares de secuencias de 1 elemento para formar secuencias ordenadas de longitud 2, fusionar pares de secuencias de longitud 2 para formar secuencias ordenadas de longitud 4, y así sucesivamente, hasta que dos secuencias de longitud  $n=2$  se fusionen para formar la secuencia ordenada final de longitud n.

### 2.3.2 Análisis de algoritmos de divide y vencerás

Cuando un algoritmo contiene una llamada recursiva a sí mismo, a menudo podemos describir su tiempo de ejecución mediante una ecuación de recurrencia o recurrencia, que describe el tiempo de ejecución general en un problema de tamaño n en términos del tiempo de ejecución en entradas más pequeñas. Luego podemos usar herramientas matemáticas para resolver la recurrencia y proporcionar límites en el rendimiento del algoritmo.

<sup>7</sup> Veremos en el Capítulo 3 cómo interpretar formalmente ecuaciones que contienen notación  $,$ .

<sup>8</sup> La expresión dxe denota el menor entero mayor o igual que x, y bxc denota el mayor entero menor o igual que x. Estas notaciones se definen en el Capítulo 3. La forma más fácil de verificar que establecer q en bp C r/ =2c produce subarreglos AŒp :: q y AŒq C 1::r de tamaños  $d_n=2e$  y  $b_n=2c$ , respectivamente, es examinar los cuatro casos que surgen dependiendo de si p y r son impares o inclusivo.

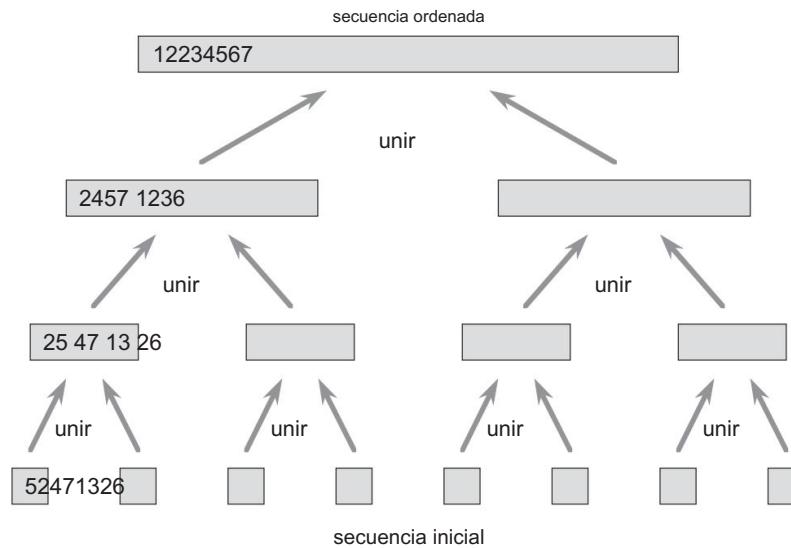


Figura 2.4 La operación de clasificación por fusión en el arreglo AD h5; 2; 4; 7; 1; 3; 2; 6i. Las longitudes de las secuencias ordenadas que se fusionan aumentan a medida que el algoritmo avanza de abajo hacia arriba.

Una recurrencia para el tiempo de ejecución de un algoritmo divide y vencerás cae fuera de los tres pasos del paradigma básico. Como antes, dejamos que  $T .n/$  sea el tiempo de ejecución de un problema de tamaño  $n$ . Si el tamaño del problema es lo suficientemente pequeño, digamos  $n$  para alguna  $c$  constante, la solución directa toma un tiempo constante, que escribimos como  $.1/$ . Suponga que nuestra división del problema produce subproblemas, cada uno de los cuales es  $1=b$  del tamaño del original. (Para el ordenamiento por combinación, tanto  $a$  como  $b$  son 2, pero veremos muchos algoritmos de divide y vencerás en los que  $a \geq b$ ). Se necesita tiempo  $T .n=b/$  para resolver un subproblema de tamaño  $n=b$ , y por lo que toma tiempo  $aT .n=b/$  para resolver uno de ellos. Si tomamos  $D_n/$  tiempo para dividir el problema en subproblemas y  $C_n/$  tiempo para combinar las soluciones de los subproblemas en la solución del problema original, obtenemos la recurrencia

si  $n=c$  ;

$T .n/ = D (.1/aT .n=b/ + C D_n/ + C C_n/)$  de lo contrario:

En el Capítulo 4, veremos cómo resolver recurrencias comunes de esta forma.

#### Análisis de clasificación por fusión

Aunque el pseudocódigo para MERGE-SORT funciona correctamente cuando el número de elementos no es par, nuestro análisis basado en recurrencia se simplifica si asumimos que

el tamaño del problema original es una potencia de 2. Cada paso de división produce dos subsecuencias de tamaño exactamente  $n=2$ . En el Capítulo 4, veremos que esta suposición no afecta el orden de crecimiento de la solución a la recurrencia.

Razonamos de la siguiente manera para configurar la recurrencia para  $T .n/$ , el tiempo de ejecución del peor de los casos de clasificación por fusión en  $n$  números. Combinar la ordenación en un solo elemento lleva un tiempo constante. Cuando tenemos  $n>1$  elementos, desglosamos el tiempo de ejecución de la siguiente manera.

Dividir: el paso de división solo calcula la mitad del subarreglo, lo que lleva un tiempo constante.

Así,  $Dn/ D ,.1/$ .

Conquistar: Resolvemos recursivamente dos subproblemas, cada uno de tamaño  $n=2$ , que contribuye  $2T .n=2/$  al tiempo de ejecución.

Combine: Ya hemos notado que el procedimiento MERGE en un elemento  $n$  el subarreglo toma el tiempo  $,n/$ , por lo que  $Cn/ D ,.n/$ .

Cuando agregamos las funciones  $Dn/$  y  $Cn/$  para el análisis de clasificación por combinación, estamos agregando una función que es  $,n/$  y una función que es  $,1/$ . Esta suma es una función lineal de  $n$ , es decir,  $,n/$ . Sumarlo al término  $2T .n=2/$  del paso de “conquistar” da la recurrencia para el tiempo de ejecución del peor de los casos  $T .n/$  del tipo de combinación:

$$\begin{aligned} & \text{si } n \leq 1 \\ T .n/ &= D ,.1/ + 2T .n=2/ + C ,n/ \text{ si } n>1 \end{aligned} \quad (2.1)$$

En el Capítulo 4, veremos el “teorema maestro”, que podemos usar para mostrar que  $T .n/$  es  $,n \lg n/$ , donde  $\lg n$  representa  $\log_2 n$ . Debido a que la función logarítmica crece más lentamente que cualquier función lineal, para entradas lo suficientemente grandes, la ordenación por fusión, con su tiempo de ejecución  $,n \lg n/$ , supera a la ordenación por inserción, cuyo tiempo de ejecución es  $,n^2/$ , en el peor de los casos.

No necesitamos el teorema maestro para entender intuitivamente por qué la solución a la recurrencia (2.1) es  $T .n/ = D ,n \lg n/$ . Reescribamos la recurrencia (2.1) como

$$\begin{aligned} & \text{si } n \leq 1 \\ T .n/ &= D ,c + 2T .n=2/ + C cn \text{ si } n>1 \end{aligned} \quad (2.2)$$

donde la constante  $c$  representa el tiempo requerido para resolver problemas de tamaño 1 así como el tiempo por elemento del arreglo de los pasos de dividir y combinar.<sup>9</sup>

<sup>9</sup>Es poco probable que la misma constante represente exactamente tanto el tiempo para resolver problemas de tamaño 1 como el tiempo por elemento del arreglo de los pasos de dividir y combinar. Podemos sortear este problema haciendo que  $c$  sea el mayor de estos tiempos y entendiendo que nuestra recurrencia da un límite superior en el tiempo de ejecución, o dejando que  $c$  sea el menor de estos tiempos y entendiendo que nuestra recurrencia da un límite inferior en el tiempo de ejecución. Ambos límites son del orden de  $n \lg n$  y, tomados en conjunto, dan un tiempo de ejecución  $,n \lg n/$ .

La Figura 2.5 muestra cómo podemos resolver la recurrencia (2.2). Por conveniencia, suponemos que  $n$  es una potencia exacta de 2. La parte (a) de la figura muestra  $T_{n/2}$ , que desarrollamos en la parte (b) en un árbol equivalente que representa la recurrencia. El término  $c_n$  es la raíz (el costo incurrido en el nivel superior de recursión), y los dos subárboles de la raíz son las dos recurrencias más pequeñas  $T_{n/2}$ . La parte (c) muestra este proceso llevado un paso más allá al expandir  $T_{n/4}$ . El costo incurrido en cada uno de los dos subnodos en el segundo nivel de recursividad es  $c_n/2$ . Continuamos expandiendo cada nodo en el árbol dividiéndolo en sus partes constituyentes según lo determine la recurrencia, hasta que los tamaños del problema se reduzcan a 1, cada uno con un costo de  $c$ . La parte (d) muestra el árbol de recursión resultante.

A continuación, sumamos los costos en cada nivel del árbol. El nivel superior tiene un costo total  $c_n$ , el siguiente nivel hacia abajo tiene un costo total  $c_n/2 + c_n/2 = c_n$ , el nivel posterior tiene un costo total  $c_n/4 + c_n/4 + c_n/4 + c_n/4 = c_n$ , y así sucesivamente. En general, el nivel  $i$  por debajo de la parte superior tiene  $2^i$  nodos, cada uno de los cuales contribuye con un costo de  $c_n/2^i$ , por lo que el  $i$ -ésimo nivel por debajo de la parte superior tiene un costo total de  $2^i c_n/2^i = c_n$ . El nivel inferior tiene  $n$  nodos, cada uno de los cuales contribuye con un costo de  $c$ , por un costo total de  $c_n$ .

El número total de niveles del árbol de recurrencia en la Figura 2.5 es  $\lg n$ , donde  $n$  es el número de hojas, correspondiente al tamaño de entrada. Un argumento inductivo informal justifica esta afirmación. El caso base ocurre cuando  $n = 1$ , en cuyo caso el árbol tiene un solo nivel. Como  $\lg 1 = 0$ , tenemos que  $\lg n$  da el número correcto de niveles. Ahora supongamos como hipótesis inductiva que el número de niveles de un árbol recursivo con  $2^i$  hojas es  $\lg 2^i = i$ . Ya que para cualquier valor de  $i$ , tenemos que  $\lg 2^{i+1} = i+1$ . Debido a que asumimos que el tamaño de entrada es una potencia de 2, el siguiente tamaño de entrada a considerar es  $2^{i+1}$ . Un árbol con  $n = 2^{i+1}$  hojas tiene un nivel más que un árbol con  $2^i$  hojas, por lo que el número total de niveles es  $i + 1$ .

Para calcular el costo total representado por la recurrencia (2.2), simplemente sumamos los costos de todos los niveles. El árbol de recurrencia tiene  $\lg n$  niveles, cada uno con un costo  $c_n$ , para un costo total de  $c_n \lg n$ . Ignorando el término de orden inferior y la constante  $c$  da el resultado deseado de  $n \lg n$ .

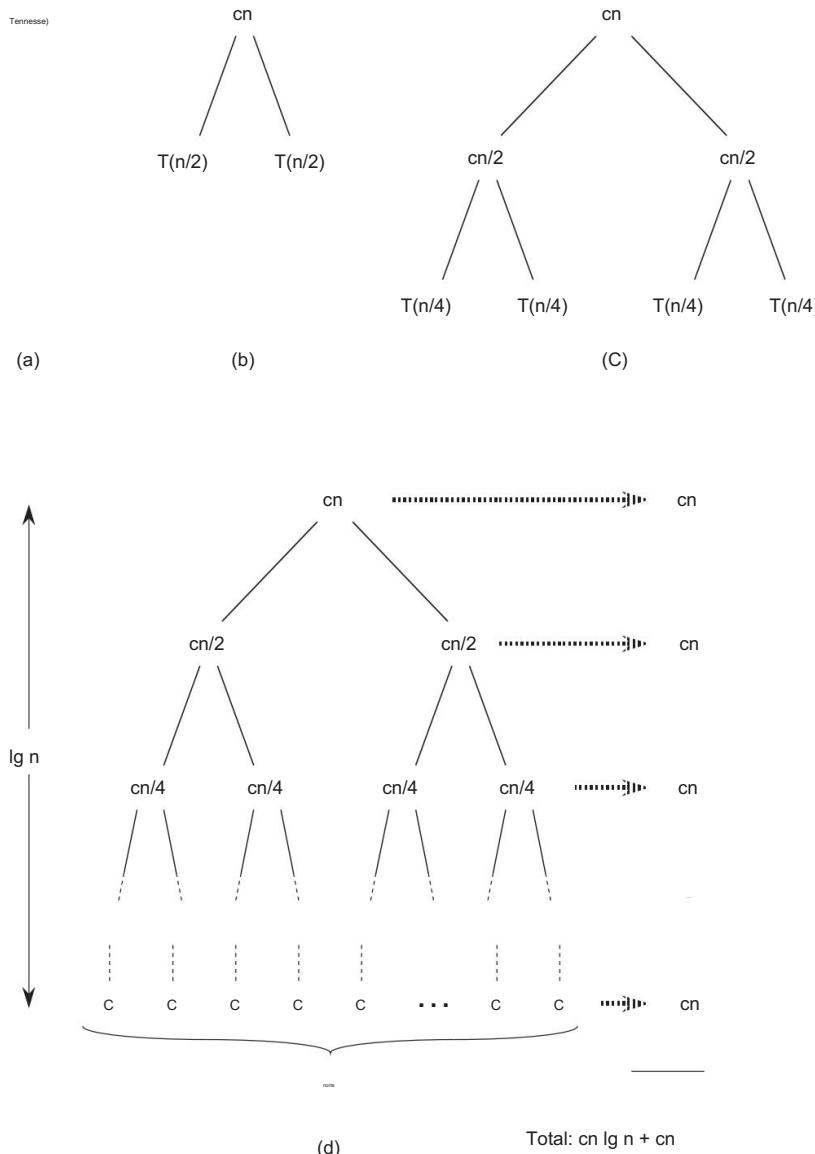
### Ejercicios

#### 2.3-1

Utilizando la figura 2.4 como modelo, ilustre la operación de clasificación por fusión en el arreglo AD h3; 41; 52; 26; 38; 57; 9; 49i.

#### 2.3-2

Vuelva a escribir el procedimiento MERGE para que no use centinelas, sino que se detenga una vez que se hayan copiado todos los elementos del arreglo L o R en A y luego copie el resto del otro arreglo nuevamente en A.

Figura 2.5 Cómo construir un árbol de recurrencia  $T(n) = 2T(n/2) + cn$ .

La parte (a) muestra  $T(n)$ , que se expande progresivamente en (b)–(d) para formar el árbol de recurrencia. El árbol completamente expandido en la parte (d) tiene niveles  $\lg n$  (es decir, tiene una altura  $\lg n$ , como se indica), y cada nivel contribuye con un costo total de  $cn$ . El costo total, por lo tanto, es  $cn \lg n + cn$ , que es  $n \lg n$ .

## 2.3-3

Use la inducción matemática para mostrar que cuando  $n$  es una potencia exacta de 2, la solución de la recurrencia

$$\text{si } n \leq 2$$

$$T(n) = 2T(n/2) + C \text{ si } n > 2^k, \text{ para } k > 1$$

es  $T(n) = O(n \lg n)$ .

## 2.3-4

Podemos expresar la ordenación por inserción como un procedimiento recursivo de la siguiente manera. Para ordenar  $A[1:n]$ , ordenamos recursivamente  $A[1:n-1]$  y luego insertamos  $A[n]$  en la matriz ordenada  $A[1:n-1]$ . Escriba una recurrencia para el tiempo de ejecución de esta versión recursiva de ordenación por inserción.

## 2.3-5

Volviendo al problema de búsqueda (vea el ejercicio 2.1-3), observe que si se ordena la secuencia A, podemos verificar el punto medio de la secuencia y eliminar la mitad de la secuencia de una consideración posterior. El algoritmo de búsqueda binaria repite este procedimiento, reduciendo a la mitad el tamaño de la porción restante de la secuencia cada vez. Escriba pseudocódigo, ya sea iterativo o recursivo, para la búsqueda binaria.

Argumente que el peor tiempo de ejecución de la búsqueda binaria es  $\Theta(\lg n)$ .

## 2.3-6

Observe que el bucle while de las líneas 5 a 7 del procedimiento INSERCIÓN-ORDENACIÓN de la Sección 2.1 utiliza una búsqueda lineal para explorar (hacia atrás) a través del subarray ordenado  $A[j:n]$ . ¿Podemos utilizar una búsqueda binaria (consulte el Ejercicio 2.3-5) en su lugar, para mejorar el tiempo de ejecución general en el peor de los casos de ordenación por inserción a  $\Theta(n \lg n)$ ?

## 2.3-7 ?

Describir un algoritmo  $\Theta(n \lg n)$ -tiempo que, dado un conjunto S de n enteros y otro entero x, determine si existen o no dos elementos en S cuya suma sea exactamente x.

## Problemas

## 2-1 Ordenación por inserción en arreglos pequeños en la ordenación

por fusión Aunque la ordenación por fusión se ejecuta en el peor de los casos y la ordenación por inserción se ejecuta en el peor de los casos, los factores constantes en la ordenación por inserción pueden hacerlo más rápido en la práctica para tamaños de problemas pequeños en muchas máquinas. Por lo tanto, tiene sentido engrosar las hojas de la recursividad utilizando la ordenación por inserción dentro de la ordenación por fusión cuando

los subproblemas se vuelven lo suficientemente pequeños. Considere una modificación a la ordenación por fusión en la que  $n=k$  sublistas de longitud  $k$  se ordenan mediante la ordenación por inserción y luego se fusionan mediante el mecanismo de fusión estándar, donde  $k$  es un valor a determinar.

- a. Muestre que la ordenación por inserción puede ordenar las  $n=k$  sublistas, cada una de longitud  $k$ , en  $.nk/\lg.n=k//$  en el peor de los casos.
- b. Muestre cómo fusionar las sublistas en  $.n \lg.n=k//$  en el peor de los casos.
- C. Dado que el algoritmo modificado se ejecuta en  $.nk C n \lg.n=k//$  en el peor de los casos, ¿cuál es el valor más grande de  $k$  en función de  $n$  para el cual el algoritmo modificado tiene el mismo tiempo de ejecución que el ordenamiento por fusión estándar? , en términos de notación ,?
- d. ¿Cómo debemos elegir  $k$  en la práctica?

## 2-2 Corrección de bubblesort Bubblesort

es un algoritmo de clasificación popular, pero ineficiente. Funciona intercambiando repetidamente elementos adyacentes que están fuera de servicio.

### BUBBLESORT.A/ 1

```
para i D 1 a A:longitud 1 para j D
    A:longitud hasta i C 1 2 si A[j] < A[j+1]
        intercambiar A[j] con A[j+1]
    4
```

- a. Sea  $A_0$  la salida de BUBBLESORT.A/. Para probar que BUBBLESORT es correcto, necesitamos probar que termina y que

$$A_0 \leq 1 \quad A_0 \leq 2 \quad \dots \quad A_0 \leq n ; \quad (2.3)$$

donde  $n$  D  $A$ :longitud. Para demostrar que BUBBLESORT realmente clasifica, ¿qué más necesitamos probar?

Las siguientes dos partes demostrarán la desigualdad (2.3).

- b. Indique con precisión una invariante de ciclo para el ciclo for en las líneas 2 a 4 y demuestre que esta invariante de ciclo se cumple. Su prueba debe usar la estructura de la prueba invariante de bucle presentada en este capítulo.

- C. Usando la condición de terminación de la invariante de ciclo demostrada en la parte (b), establezca una invariante de ciclo para el ciclo for en las líneas 1 a 4 que le permitirá demostrar la igualdad (2.3). Su prueba debe usar la estructura de la prueba invariante de bucle presentada en este capítulo.

- d. ¿Cuál es el peor tiempo de ejecución de bubblesort? ¿Cómo se compara con el tiempo de ejecución del tipo de inserción?

### 2-3 Corrección de la regla de Horner

El siguiente fragmento de código implementa la regla de Horner para evaluar un polinomio

```
P .x/ D Xn      akxk
          kD0
D a0 C x.a1 C x.a2 CC x.an1 C xan// ;
```

dados los coeficientes  $a_0; a_1; \dots; a_n$  y un valor para  $x$ :

1 y D 0 2  
para i D n hasta 0 3 y D ai  
C xy

- a. En términos de notación,, ¿cuál es el tiempo de ejecución de este fragmento de código para la Regla de Horner?
- b. Escriba pseudocódigo para implementar el algoritmo ingenuo de evaluación de polinomios que calcula cada término del polinomio desde cero. ¿Cuál es el tiempo de ejecución de este algoritmo? ¿Cómo se compara con la regla de Horner?

C. Considere el siguiente bucle invariante:

Al comienzo de cada iteración del bucle for de las líneas 2 y 3,

```
... .iC1/
y D   x     akCiC1xk :
          kD0
```

Interprete una sumatoria sin términos como igual a 0. Siguiendo la estructura de la prueba invariante de bucles presentada en este capítulo, use esta prueba invariante de bucles para demostrar que, al final, y D<sub>kD0</sub> akxk.

- d. Concluya argumentando que el fragmento de código dado evalúa correctamente un polinomial caracterizado por los coeficientes  $a_0; a_1; \dots; a_n$ .

### 2-4 Inversiones

Sea  $A \in \mathbb{R}^n$  un arreglo de  $n$  números distintos. Si  $i < j$  y  $A[i] > A[j]$ , par  $i; j$  se llama entonces una inversión de  $A$ .

- a. Haz una lista de las cinco inversiones del arreglo  $h = [2; 3; 8; 6; 1]$ .

- b. Qué matriz con elementos del conjunto  $f1; 2; \dots; n$  tiene la mayor cantidad de inversiones?  
 ¿Cuántos tiene?
- C. ¿Cuál es la relación entre el tiempo de ejecución de la ordenación por inserción y el número de inversiones en la matriz de entrada? Justifica tu respuesta.
- d. Proporcione un algoritmo que determine el número de inversiones en cualquier permutación en  $n$  elementos en  $,n \lg n/$  en el peor de los casos. (Sugerencia: modifique la ordenación por fusión).

### Notas del capítulo

En 1968, Knuth publicó el primero de tres volúmenes con el título general *The Art of Computer Programming* [209, 210, 211]. El primer volumen marcó el comienzo del estudio moderno de los algoritmos informáticos con un enfoque en el análisis del tiempo de ejecución, y la serie completa sigue siendo una referencia atractiva y valiosa para muchos de los temas presentados aquí. Según Knuth, la palabra “algoritmo” se deriva del nombre “al-Khow^arizm^”, un matemático persa del siglo IX.

Aho, Hopcroft y Ullman [5] defendieron el análisis asintótico de los algoritmos, utilizando las notaciones que se presentan en el capítulo 3, incluida la notación  $\sim$ , como un medio para comparar el rendimiento relativo. También popularizaron el uso de relaciones de recurrencia para describir los tiempos de ejecución de los algoritmos recursivos.

Knuth [211] proporciona un tratamiento enciclopédico de muchos algoritmos de clasificación. Su comparación de algoritmos de clasificación (página 381) incluye análisis exactos de conteo de pasos, como el que realizamos aquí para la clasificación por inserción. La discusión de Knuth sobre el ordenamiento por inserción abarca varias variaciones del algoritmo. La más importante de ellas es la ordenación de Shell, introducida por DL Shell, que utiliza la ordenación por inserción en subsecuencias periódicas de la entrada para producir un algoritmo de ordenación más rápido.

Knuth también describe la ordenación por combinación. Menciona que en 1938 se inventó un clasificador mecánico capaz de fusionar dos mazos de tarjetas perforadas en una sola pasada. J. von Neumann, uno de los pioneros de la informática, aparentemente escribió un programa para clasificar por fusión en la computadora EDVAC en 1945. .

Gries [153] describe la historia temprana de la demostración de programas correctos y atribuye a P. Naur el primer artículo en este campo. Gries atribuye invariantes de bucle a RW Floyd. El libro de texto de Mitchell [256] describe el progreso más reciente en la prueba de que los programas son correctos.

---

## 3 Crecimiento de Funciones

El orden de crecimiento del tiempo de ejecución de un algoritmo, definido en el Capítulo 2, brinda una caracterización simple de la eficiencia del algoritmo y también nos permite comparar el desempeño relativo de algoritmos alternativos. Una vez que el tamaño de entrada  $n$  sea lo suficientemente grande, la ordenación por fusión, con su tiempo de ejecución en el peor de los casos  $.n \lg n/$ , supera a la ordenación por inserción, cuyo tiempo de ejecución en el peor de los casos es  $.n^2/$ . Aunque a veces podemos determinar el tiempo de ejecución exacto de un algoritmo, como hicimos con la ordenación por inserción en el Capítulo 2, la precisión adicional no suele valer el esfuerzo de calcularla. Para entradas lo suficientemente grandes, las constantes multiplicativas y los términos de orden inferior de un tiempo de ejecución exacto están dominados por los efectos del tamaño de la entrada en sí.

Cuando observamos tamaños de entrada lo suficientemente grandes como para que solo sea relevante el orden de crecimiento del tiempo de ejecución, estamos estudiando la eficiencia asintótica de los algoritmos. Es decir, nos preocupa cómo aumenta el tiempo de ejecución de un algoritmo con el tamaño de la entrada en el límite, ya que el tamaño de la entrada aumenta sin límites.

Por lo general, un algoritmo que sea asintóticamente más eficiente será la mejor opción para todas las entradas, excepto para las muy pequeñas.

Este capítulo proporciona varios métodos estándar para simplificar el análisis asintótico de algoritmos. La siguiente sección comienza definiendo varios tipos de "notación asintótica", de los cuales ya hemos visto un ejemplo en la notación  $\dots$ . Luego presentamos varias convenciones de notación utilizadas a lo largo de este libro y, finalmente, revisamos el comportamiento de las funciones que comúnmente surgen en el análisis de algoritmos.

---

### 3.1 Notación asintótica

Las notaciones que usamos para describir el tiempo de ejecución asintótico de un algoritmo se definen en términos de funciones cuyos dominios son el conjunto de números naturales  $\mathbb{N} = \{0; 1; 2; \dots\}$ . Tales notaciones son convenientes para describir la función de tiempo de ejecución del peor de los casos  $T(n)$ , que generalmente se define solo en tamaños de entrada enteros. A veces encontramos conveniente, sin embargo, abusar de la notación asintótica en una variedad de

variedad de caminos. Por ejemplo, podríamos extender la notación al dominio de los números reales o, alternativamente, restringirla a un subconjunto de los números naturales. Sin embargo, debemos asegurarnos de entender el significado preciso de la notación para que cuando abusemos, no la usemos mal . Esta sección define las notaciones asintóticas básicas y también presenta algunos abusos comunes.

#### Notación asintótica, funciones y tiempos de ejecución

Usaremos la notación asintótica principalmente para describir los tiempos de ejecución de los algoritmos, como cuando escribimos que el tiempo de ejecución del peor caso de la ordenación por inserción es  $.n^2/$ . Sin embargo, la notación asintótica en realidad se aplica a las funciones. Recuerde que caracterizamos el tiempo de ejecución del peor de los casos de ordenación por inserción como  $a n^2 C b n C c$ , para algunas constantes  $a$ ,  $b$  y  $c$ . Al escribir que el tiempo de ejecución de la ordenación por inserción es  $.n^2/$ , abstrajimos algunos detalles de esta función. Debido a que la notación asintótica se aplica a las funciones, lo que estábamos escribiendo como  $.n^2/$  era la función  $a n^2 C b n C c$ , que en ese caso caracterizaba el peor tiempo de ejecución del tipo de inserción.

En este libro, las funciones a las que aplicamos la notación asintótica generalmente caracterizarán los tiempos de ejecución de los algoritmos. Pero la notación asintótica puede aplicarse a funciones que caracterizan algún otro aspecto de los algoritmos (la cantidad de espacio que utilizan, por ejemplo), o incluso a funciones que no tienen nada que ver con los algoritmos.

Incluso cuando usamos la notación asintótica para aplicarla al tiempo de ejecución de un algoritmo, necesitamos entender a qué tiempo de ejecución nos referimos. A veces nos interesa el peor tiempo de ejecución. A menudo, sin embargo, deseamos caracterizar el tiempo de ejecución sin importar la entrada. En otras palabras, a menudo deseamos hacer una declaración general que cubra todas las entradas, no solo el peor de los casos. Veremos notaciones asintóticas que son muy adecuadas para caracterizar los tiempos de ejecución sin importar la entrada.

#### $.-$ notación

En el Capítulo 2, encontramos que el tiempo de ejecución del ordenamiento por inserción en el peor de los casos es  $T .n/ D .n^2/$ . Definamos qué significa esta notación. Para una función dada  $g(n)$ , denotamos por  $.g(n) /$  el conjunto de funciones

$.g(n) / D f(n) / W$  existen constantes positivas  $c_1$ ,  $c_2$  y  $n_0$  tales que  $0 < c_1 g(n) / f(n) / c_2 g(n)$  para todo  $n > n_0$  :

1

---

<sup>1</sup>Dentro de la notación de conjuntos, los dos puntos significan "tal que".

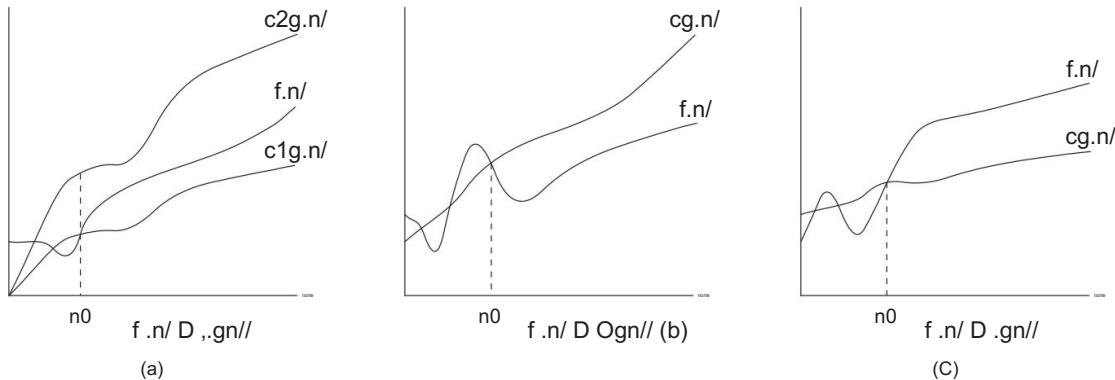


Figura 3.1 Ejemplos gráficos de las notaciones  $\in$ ,  $O$  y  $\in$ . En cada parte, el valor de  $n_0$  mostrado es el valor mínimo posible; cualquier valor mayor también funcionaría. (a) La notación  $\in$ , limita una función dentro de factores constantes. Escribimos  $f .n/ \in D .gn//$  si existen constantes positivas  $n_0$ ,  $c_1$  y  $c_2$  tales que a la derecha de  $n_0$ , el valor de  $f .n/$  siempre se encuentra entre  $c_1g.n/$  y  $c_2g .n/$  inclusive. (b) La notación  $O$  da un límite superior para una función dentro de un factor constante. Escribimos  $f .n/ \in Ogn//$  si hay constantes positivas  $n_0$  y  $c$  tales que a la derecha de  $n_0$ , el valor de  $f .n/$  siempre se encuentra en o por debajo de  $cg.n/$ . (c) La notación  $\in$  da un límite inferior para una función dentro de un factor constante. Escribimos  $f .n/ \in D .gn//$  si hay constantes positivas  $n_0$  y  $c$  tales que a la derecha de  $n_0$ , el valor de  $f .n/$  siempre se encuentra en o por encima de  $cg.n/$ .

Una función  $f .n/$  pertenece al conjunto  $.gn//$  si existen constantes positivas  $c_1$  y  $c_2$  tales que puede ser “emparedada” entre  $c_1g.n/$  y  $c_2g.n/$ , para  $n$  suficientemente grande. Como  $.gn//$  es un conjunto, podríamos escribir “ $f .n/ \in 2 .gn//$ ” para indicar que  $f .n/$  es miembro de  $.gn//$ . En su lugar, normalmente escribiremos “ $f .n/ \in D .gn//$ ” para expresar la misma noción. Es posible que se sienta confundido porque abusamos de la igualdad de esta manera, pero veremos más adelante en esta sección que hacerlo tiene sus ventajas.

La figura 3.1(a) da una imagen intuitiva de las funciones  $f .n/ \in gn/$ , donde  $f .n/ \in D .gn//$ . Para todos los valores de  $n$  a la derecha de  $n_0$ , el valor de  $f .n/$  se encuentra en o por encima de  $c_1g.n/$  y en o por debajo de  $c_2g.n/$ . En otras palabras, para todo  $n > n_0$ , la función  $f .n/$  es igual a  $gn/$  dentro de un factor constante. Decimos que  $gn/$  es un límite asintóticamente estrecho para  $f .n/$ .

La definición de  $.gn//$  requiere que todo miembro  $f .n/ \in 2 .gn//$  sea asintóticamente no negativo, es decir, que  $f .n/$  sea no negativo siempre que  $n$  sea suficientemente grande. (Una función asintóticamente positiva es aquella que es positiva para todo  $n$  suficientemente grande). En consecuencia, la función  $gn/$  en sí misma debe ser asintóticamente no negativa, o de lo contrario el conjunto  $.gn//$  está vacío. Por lo tanto, supondremos que cada función utilizada dentro de la notación  $\in$ , es asintóticamente no negativa. Esta suposición es válida también para las otras notaciones asintóticas definidas en este capítulo.

En el Capítulo 2, presentamos una noción informal de notación , que equivalía a desechar los términos de orden inferior e ignorar el coeficiente principal del término de orden superior.

Justifiquemos brevemente esta intuición usando la definición formal para mostrar que

$\frac{1}{2}n^2 \leq 3n^2$ . Para hacerlo, debemos determinar positivo constantes  $c_1, c_2$  y  $n_0$  tales que

$$c_1 n^2 \leq \frac{1}{3}n^2 \leq c_2 n^2$$

para todo  $n > n_0$ . Dividiendo por  $n^2$  se obtiene

$$c_1 \leq \frac{1}{3} \leq c_2 : \quad \text{norte}$$

Podemos hacer que la desigualdad de la derecha se cumpla para cualquier valor de  $n$  eligiendo izquierda se cualquier  $c_1 = \frac{1}{2}$ . Del mismo modo, podemos hacer que la desigualdad de la cumpla para cualquier valor  $c_2$  constante de  $n$  eligiendo cualquier constante  $c_2 = 3$ . Así, Ciertamente, otras  $c_2 \geq \frac{1}{2}$ , y  $n_0 \geq 7$ , podemos eligiendo  $c_2 = 7$ ,  $n_0 = 14$ ,  $c_1 = \frac{1}{2}$ ,  $n^2 \leq 3n^2$ . verificar que existen elecciones para las constantes, pero lo importante es que existe alguna diferente Tenga en cuenta que estas constantes dependen elección.  $n^2 \leq 3n^2$ ; una función de la función que pertenece a  $n^2$  normalmente requeriría constantes diferentes.

También podemos usar la definición formal para verificar que  $6n^3 \leq n^2$ . Supongamos, con fines de contradicción, que  $c_2 < 6$  existen de manera que  $6n^3 > c_2 n^2$  para todo  $n > n_0$ . Pero luego, al dividir por  $n^2$ , se obtiene  $n > c_2 = 6$ , lo que posiblemente no se cumpla para  $n$  arbitrariamente grande, ya que  $c_2$  es constante.

Intuitivamente, los términos de orden inferior de una función asintóticamente positiva pueden ignorarse al determinar los límites asintóticamente estrechos porque son insignificantes para  $n$  grande. Cuando  $n$  es grande, incluso una pequeña fracción del término de mayor orden es suficiente para dominar los términos de menor orden. Por lo tanto, establecer  $c_1$  en un valor ligeramente menor que el coeficiente del término de mayor orden y establecer  $c_2$  en un valor ligeramente mayor permite satisfacer las desigualdades en la definición de la notación . El coeficiente del término de mayor orden también se puede ignorar, ya que solo cambia  $c_1$  y  $c_2$  por un factor constante igual al coeficiente.

Como ejemplo, considere cualquier función cuadrática  $f(n) = an^2 + bn + c$ , donde  $a, b$  y  $c$  son constantes y  $a > 0$ . Descartando los términos de orden inferior e ignorando la constante se obtiene  $f(n) \leq an^2$ . Formalmente, para mostrar lo mismo, tomamos las constantes  $c_1 = a$ ,  $c_2 = 7a$  y  $n_0 = 2 \max\{|b|, |c|\} = a$ . Puede verificar que  $0 < c_1 n^2 \leq an^2 \leq bn + c \leq c_2 n^2$  para todo  $n > n_0$ . En general, para cualquier polinomio  $p(n)$  de grado  $d$  a  $n^d$ , donde los  $a_i$  son constantes y  $a_d > 0$ , tenemos  $p(n) \leq n^d$  (vea el Problema 3-1).

Dado que cualquier constante es un polinomio de grado 0, podemos expresar cualquier función constante como  $n^0$  o  $1$ . Esta última notación es un abuso menor, sin embargo, porque el

expresión no indica qué variable tiende a infinito.<sup>2</sup> A menudo usaremos la notación „1/ para indicar una función constante o constante con respecto a alguna variable.

### notación O

La notación , limita asintóticamente una función desde arriba y desde abajo. Cuando solo tenemos un límite superior asintótico, usamos la notación O. Para una función dada  $g_n$ , denotamos por  $O(g)$  (pronunciado “gran-oh de  $g$  de  $n$ ” o a veces simplemente “oh de  $g$  de  $n$ ”) el conjunto de funciones

$O(g) = \{f(n) : \exists c, n_0 \in \mathbb{N} \text{ such that } 0 < f(n) < cg(n) \text{ for all } n \geq n_0\}$

Usamos la notación O para dar un límite superior a una función, dentro de un factor constante. La figura 3.1(b) muestra la intuición detrás de la notación O. Para todos los valores  $n$  a la derecha de  $n_0$ , el valor de la función  $f(n)$  está en o por debajo de  $cg(n)$ .

Escribimos  $f(n) \in O(g)$  para indicar que una función  $f(n)$  es miembro del conjunto  $O(g)$ . Tenga en cuenta que  $f(n) \in O(g)$  implica  $f(n) \in O(g')$ , ya que la notación „1/ es una noción más fuerte que la notación O. Escrito en teoría de conjuntos, tenemos  $O(g) \subseteq O(g')$ . Por lo tanto, nuestra prueba de que cualquier función cuadrática  $a^2 n^2 + bn + c$ , donde  $a > 0$ , está en  $O(n^2)$  también muestra que cualquier función cuadrática está en  $O(n^2)$ .

Lo que puede ser más sorprendente es que cuando  $a > 0$ , cualquier función lineal  $an + b$  está en  $O(n^2)$ , lo cual se verifica fácilmente tomando  $c = D \max(1, |b/a|)$ .

Si ha visto la notación O antes, puede que le resulte extraño que escribamos, por ejemplo,  $n \in O(n^2)$ . En la literatura, a veces encontramos notación O que describe informalmente límites estrechos asintóticamente, es decir, lo que hemos definido usando notación „1/. En este libro, sin embargo, cuando escribimos  $f(n) \in O(g)$ , simplemente afirmamos que algún múltiplo constante de  $g(n)$  es un límite superior asintótico en  $f(n)$ , sin afirmar qué tan ajustado es el límite superior. es. Distinguir los límites superiores asintóticos de los límites estrechos asintóticamente es estándar en la literatura de algoritmos.

Usando la notación O, a menudo podemos describir el tiempo de ejecución de un algoritmo simplemente inspeccionando la estructura general del algoritmo. Por ejemplo, la estructura de bucle doblemente anidado del algoritmo de clasificación por inserción del Capítulo 2 produce inmediatamente un límite superior  $O(n^2)$  en el tiempo de ejecución del peor de los casos: el costo de cada iteración del bucle interno está acotado desde arriba por  $O(1)$  (constante), los índices i

<sup>2</sup>El verdadero problema es que nuestra notación ordinaria para funciones no distingue funciones de valores. En cálculo, los parámetros de una función están claramente especificados: la función  $n^2$  podría escribirse como  $n:n^2$ , o incluso  $r:r^2$ . Sin embargo, adoptar una notación más rigurosa complicaría las manipulaciones algebraicas, por lo que elegimos tolerar el abuso.

y j son como máximo n, y el ciclo interno se ejecuta como máximo una vez para cada uno de los n2 pares de valores para i y j .

Dado que la notación O describe un límite superior, cuando la usamos para limitar el tiempo de ejecución de un algoritmo en el peor de los casos, tenemos un límite en el tiempo de ejecución del algoritmo en cada entrada: la declaración general que discutimos anteriormente. Por lo tanto, el límite O.n2/ en el peor de los casos en el tiempo de ejecución de la ordenación por inserción también se aplica a su tiempo de ejecución en cada entrada. Sin embargo , el límite „n2/ en el tiempo de ejecución de ordenación por inserción en el peor de los casos no implica un límite „n2/ en el tiempo de ejecución de ordenación por inserción en cada entrada. Por ejemplo, vimos en el Capítulo 2 que cuando la entrada ya está ordenada, la ordenación por inserción se ejecuta en tiempo „n/.

Técnicamente, es un abuso decir que el tiempo de ejecución del ordenamiento por inserción es O.n2/, ya que para un n dado, el tiempo de ejecución real varía, dependiendo de la entrada particular de tamaño n. Cuando decimos "el tiempo de ejecución es O.n2/", queremos decir que hay una función f .n/ que es O.n2/ tal que para cualquier valor de n, sin importar qué entrada particular de tamaño n se elija, el tiempo de ejecución en esa entrada está acotado desde arriba por el valor f .n/. De manera equivalente, queremos decir que el tiempo de ejecución en el peor de los casos es O.n2/.

#### -notación

Así como la notación O proporciona un límite superior asintótico en una función, la notación proporciona un límite inferior asintótico. Para una función dada gn/, denotamos por .gn// (pronunciado “big-omega of g of n” o a veces simplemente “omega of g of n”) el conjunto de funciones

.gn// D ff .n/ W existen constantes positivas c y n0 tales que  

$$0 \leq c g(n) \leq f(n) \text{ para todo } n \geq n_0$$

La figura 3.1(c) muestra la intuición detrás de la notación. Para todos los valores n en o a la derecha de n0, el valor de f .n/ está en o por encima de cg.n/.

A partir de las definiciones de las notaciones asintóticas que hemos visto hasta ahora, es fácil probar el siguiente teorema importante (vea el ejercicio 3.1-5).

#### Teorema 3.1

Para dos funciones cualesquiera f .n/ y gn/, tenemos f .n/ D „gn// si y sólo si f .n/ D Ogn// y f .n/ D .gn//.

Como ejemplo de la aplicación de este teorema, nuestra prueba de que an2 C bn C c D „n2/ para cualquier constante a, b y c, donde a>0, implica inmediatamente que an2 C bn C c D „n2/ y an2 CbnCc D O.n2/. En la práctica, en lugar de usar el Teorema 3.1 para obtener límites asintóticos superior e inferior a partir de límites asintóticamente ajustados, como hicimos en este ejemplo, generalmente lo usamos para probar límites asintóticamente ajustados a partir de límites asintóticos superior e inferior.

Cuando decimos que el tiempo de ejecución (sin modificador) de un algoritmo es  $.gn//$ , queremos decir que no importa qué entrada particular de tamaño  $n$  se elija para cada valor de  $n$ , el tiempo de ejecución en esa entrada es al menos una constante veces  $gn//$ , para  $n$  suficientemente grande. De manera equivalente, estamos dando un límite inferior en el mejor tiempo de ejecución de un algoritmo. Por ejemplo, en el mejor de los casos, el tiempo de ejecución de la ordenación por inserción es  $.n//$ , lo que implica que el tiempo de ejecución de la ordenación por inserción es  $.n//$ .

Por lo tanto, el tiempo de ejecución del ordenamiento por inserción pertenece tanto a  $.n//$  como a  $O.n2//$ , ya que se encuentra entre una función lineal de  $n$  y una función cuadrática de  $n$ .

Además, estos límites son asintóticamente tan estrechos como sea posible: por ejemplo, el tiempo de ejecución de la ordenación por inserción no es  $.n2//$ , ya que existe una entrada para la cual la ordenación por inserción se ejecuta en el tiempo  $.n//$  (por ejemplo, cuando la entrada ya está ordenada). Sin embargo, no es contradictorio decir que el peor tiempo de ejecución del ordenamiento por inserción es  $.n2//$ , ya que existe una entrada que hace que el algoritmo tome  $.n2//$  tiempo.

### Notación asintótica en ecuaciones y desigualdades

Ya hemos visto cómo se puede usar la notación asintótica dentro de las fórmulas matemáticas. Por ejemplo, al introducir la notación  $O$ , escribimos " $n D O.n2//$ ". También podríamos escribir  $2n2 C 3nC1 D 2n2 C.n//$ . ¿Cómo interpretamos tales fórmulas?

Cuando la notación asintótica está sola (es decir, no dentro de una fórmula más grande) en el lado derecho de una ecuación (o desigualdad), como en  $n D O.n2//$ , ya hemos definido que el signo igual significa pertenencia a un conjunto :  $n \in O.n2//$ . En general, sin embargo, cuando la notación asintótica aparece en una fórmula, la interpretamos como si representara alguna función anónima que no nos importa nombrar. Por ejemplo, la fórmula  $2n2 C 3n C 1 D 2n2 C .n//$  significa que  $2n2 C 3n C 1 D 2n2 C f .n//$ , donde  $f .n//$  es alguna función en el conjunto  $.n//$ . En este caso, hacemos  $f .n// D 3n C 1$ , que de hecho está en  $.n//$ .

El uso de la notación asintótica de esta manera puede ayudar a eliminar los detalles no esenciales y el desorden en una ecuación. Por ejemplo, en el Capítulo 2 expresamos el peor tiempo de ejecución del ordenamiento por fusión como la recurrencia

$$T .n// D 2T .n=2// C .n// :$$

Si sólo estamos interesados en el comportamiento asintótico de  $T .n//$ , no tiene sentido especificar exactamente todos los términos de orden inferior; todos ellos se entienden incluidos en la función anónima denotada por el término  $.n//$ .

Se entiende que el número de funciones anónimas en una expresión es igual al número de veces que aparece la notación asintótica. Por ejemplo, en la expresión

solo hay una única función anónima (una función de  $i$ ). Por lo tanto, esta expresión no es lo mismo que  $O.1/ \leq O.2/ \leq CC On/$ , que en realidad no tiene una interpretación clara.

En algunos casos, la notación asintótica aparece en el lado izquierdo de una ecuación, como en

$2n^2 \leq C \cdot n / D \cdot n^2 / :$

Interpretamos tales ecuaciones usando la siguiente regla: no importa cómo se elijan las funciones anónimas a la izquierda del signo igual, hay una forma de elegir las funciones anónimas a la derecha del signo igual para que la ecuación sea válida.

Así, nuestro ejemplo significa que para cualquier función  $f \cdot n / 2 \cdot n /$ , existe alguna función  $gn / 2 \cdot n^2 /$  tal que  $2n^2 \leq C f \cdot n / D gn /$  para todo  $n$ . En otras palabras, el lado derecho de una ecuación proporciona un nivel de detalle más bajo que el lado izquierdo.

Podemos encadenar varias de tales relaciones, como en

$2n^2 \leq C 3n \leq C 1 D 2n^2 \leq C \cdot n / D \cdot n^2 / :$

Podemos interpretar cada ecuación por separado según las reglas anteriores. La primera ecuación dice que existe alguna función  $f \cdot n / 2 \cdot n /$  tal que  $2n^2 \leq C 3n \leq C 1 D 2n^2 \leq C f \cdot n /$  para todo  $n$ . La segunda ecuación dice que para cualquier función  $gn / 2 \cdot n /$  (como la  $f \cdot n /$  recién mencionada), existe alguna función  $hn / 2 \cdot n^2 /$  tal que  $2n^2 \leq C gn / D hn /$  para todo  $n$ . Nótese que esta interpretación implica que  $2n^2 \leq C 3n \leq C 1 D \cdot n^2 /$ , que es lo que intuitivamente da el encadenamiento de ecuaciones

a nosotros.

notación o

El límite superior asintótico proporcionado por la notación  $O$  puede o no ser asintóticamente estrecho. El límite  $2n^2 \leq O.n^2 /$  es asintóticamente estrecho, pero el límite  $2n \leq O.n^2 /$  no lo es. Usamos la notación  $o$  para denotar un límite superior que no es asintóticamente estrecho. Definimos formalmente  $ogn//$  ("pequeño-oh de  $g$  de  $n$ ") como el conjunto

$ogn// \leq D ff \cdot n / W$  para cualquier constante positiva  $c > 0$ , existe una constante  $n_0 > 0$   
tal que  $0 f \cdot n / < cg.n /$  para todo  $n \geq n_0$  :

Por ejemplo,  $2n \leq o.n^2 /$ , pero  $2n^2 \not\leq o.n^2 /$ .

Las definiciones de notación  $O$  y notación  $o$  son similares. La principal diferencia es que en  $f \cdot n / \leq Ogn//$ , el límite  $0 f \cdot n / \leq cg.n /$  se cumple para alguna constante  $c > 0$ , pero en  $f \cdot n / \leq ogn//$ , el límite  $0 f \cdot n / < cg.n /$  vale para todas las constantes  $c > 0$ . Intuitivamente, en la notación  $o$ , la función  $f \cdot n /$  se vuelve insignificante en relación con  $gn /$  cuando  $n$  tiende a infinito; eso es,

$$\liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq 0 \quad (3.1)$$

Algunos autores utilizan este límite como definición de la notación o; la definición de este libro también restringe las funciones anónimas para que sean asintóticamente no negativas.

### !-notación

Por analogía, la notación ! es a la notación - como la notación o es a la notación O. Usamos la notación ! para denotar un límite inferior que no es asintóticamente estrecho. Una forma de definirlo es por

$f(n) \geq g(n)$  si y solo si  $\liminf_{n \rightarrow \infty} f(n) \geq g(n)$  :

Formalmente, sin embargo, definimos  $\liminf_{n \rightarrow \infty} g(n)$  ("pequeño-omega de g de n") como el conjunto  $\{g(n) : n \in \mathbb{N}\}$  para cualquier constante positiva  $c > 0$ , existe una constante  $n_0 > 0$  tal que  $cg(n) < g(n)$  para todo  $n > n_0$  : Por ejemplo,

$n^2 = 2 \leq n$ , pero  $n^2 = 2 \neq n$ . La relación  $f(n) \geq g(n)$  implica que  $f(n) \geq g(n)$

$$\liminf_{n \rightarrow \infty} f(n) \geq 1$$

si el límite existe. Es decir,  $f(n)$  se vuelve arbitrariamente grande en relación con  $g(n)$  cuando  $n$  tiende a infinito.

### Comparando funciones

Muchas de las propiedades relacionales de los números reales también se aplican a las comparaciones asintóticas. Para lo siguiente, suponga que  $f(n)$  y  $g(n)$  son asintóticamente positivas.

#### Transitividad:

$f(n) \geq g(n)$  y  $g(n) \geq h(n)$  implican  $f(n) \geq h(n)$  ;  $f(n) \geq g(n)$  y  $g(n) \geq h(n)$  implican  $f(n) \geq h(n)$  ;  $f(n) \geq g(n)$  y  $g(n) \geq h(n)$  implican  $f(n) \geq h(n)$  ;  $f(n) \geq g(n)$  y  $g(n) \geq h(n)$  implican  $f(n) \geq h(n)$  ;  $f(n) \geq g(n)$  y  $g(n) \geq h(n)$  implican  $f(n) \geq h(n)$  :

#### Reflexividad:

$f(n) \geq f(n)$  ;  $f(n) \geq f(n)$   
de  $f(n) \geq f(n)$  ;  $f(n) \geq f(n)$  :

Simetría:

$$f .n/ D ,gn// \text{ si y solo si } gn/ D ,f .n// :$$

Transponer simetría:

$$f .n/ D Ogn// \text{ si y sólo si } gn/ D .f .n// ; f .n/ D ogn// \text{ si y solo si } gn/$$

$$D !.f .n// :$$

Debido a que estas propiedades son válidas para las notaciones asintóticas, podemos establecer una analogía entre la comparación asintótica de dos funciones  $f$  y  $g$  y la comparación de dos números reales  $a$  y  $b$ :

$$\begin{aligned} f .n/ D Ogn// \text{ es como } abf .n/ D .gn// \text{ es } & : \\ \text{como } abf .n/ D ,gn// \text{ es como } D bf .n/ D & : \\ \text{ogn// } f .n/ D !.gn// \text{ es como } a>b: & : \\ \text{es como } a< b; & \end{aligned}$$

Decimos que  $f .n/$  es asintóticamente menor que  $gn/$  si  $f .n/ D ogn//$ , y  $f .n/$  es asintóticamente mayor que  $gn/$  si  $f .n/ D !.gn//$ .

Sin embargo, una propiedad de los números reales no se traslada a la notación asintótica:

Tricotomía: Para cualesquiera dos números reales  $a$  y  $b$ , exactamente uno de los siguientes debe mantener:  $a< b$ ,  $a = b$ , o  $a> b$ .

Aunque se pueden comparar dos números reales cualesquiera, no todas las funciones son asintóticamente comparables. Es decir, para dos funciones  $f .n/$  y  $gn/$ , puede darse el caso de que ni  $f .n/ D Ogn//$  ni  $f .n/ D .gn//$  se cumplan. Por ejemplo, no podemos comparar las funciones  $n$  y  $n^{1/\sqrt{n}}$  usando notación asintótica, ya que el valor del exponente en  $n^{1/\sqrt{n}}$  oscila entre 0 y 2, tomando todos los valores intermedios.

## Ejercicios

### 3.1-1

Sean  $f .n/$  y  $gn/$  funciones asintóticamente no negativas. Utilizando la definición básica de notación  $\sim$ , demuestre que  $\max(f .n/; gn// D ,f .n/ C gn//)$ .

### 3.1-2

Muestre que para cualquier constante real  $a$  y  $b$ , donde  $b>0$ ,  $.n C$

$$a/b D ,nb/ :$$

(3.2)

3.1-3

Explique por qué la afirmación "El tiempo de ejecución del algoritmo A es al menos  $O(n^2)$ " no tiene sentido.

3.1-4

¿Es  $2nC1 D O(2n)$ ? es  $22n D O(2n)$ ?

3.1-5

Demostrar el teorema 3.1.

3.1-6

Demuestre que el tiempo de ejecución de un algoritmo es  $\Omega(n)$  si y solo si su tiempo de ejecución en el peor de los casos es  $\Omega(n)$  y su tiempo de ejecución en el mejor de los casos es  $\Omega(n)$ .

3.1-7

Demuestre que  $\Omega(n) \setminus \Theta(n)$  es el conjunto vacío.

3.1-8

Podemos extender nuestra notación al caso de dos parámetros  $n$  y  $m$  que pueden llegar al infinito independientemente a diferentes velocidades. Para una función dada  $g(n; m)$ , lo denotamos por  $\Omega(g(n; m))$  el conjunto de funciones

$\Omega(g(n; m)) = \{f(n; m) \mid \exists c, n_0, m_0 \text{ tales que } \forall n > n_0, m > m_0 : f(n; m) \geq c g(n; m)\}$

Dé las definiciones correspondientes para  $\Theta(g(n; m))$  y  $\mathcal{O}(g(n; m))$ .

## 3.2 Notaciones estándar y funciones comunes

Esta sección repasa algunas funciones y notaciones matemáticas estándar y explora las relaciones entre ellos. También ilustra el uso de las notaciones asintóticas.

monotonicidad

Una función  $f(n)$  es monótonamente creciente si  $m < n$  implica  $f(m) < f(n)$ .

De manera similar, es monótonamente decreciente si  $m < n$  implica  $f(m) > f(n)$ . Una función  $f(n)$  es estrictamente creciente si  $m < n$  implica  $f(m) < f(n)$  y estrictamente decreciente si  $m < n$  implica  $f(m) > f(n)$ .

### Pisos y techos

Para cualquier número real  $x$ , denotamos el mayor entero menor o igual que  $x$  por  $\lfloor x \rfloor$  (léase “el piso de  $x$ ”) y el menor entero mayor o igual que  $x$  por  $\lceil x \rceil$  (léase “el techo de  $x$ ”). Para todo  $x$  real,

$$\lfloor x \rfloor < x < \lceil x \rceil \quad (3.3)$$

Para cualquier entero  $n$ ,

$$\lfloor n \rfloor = n \quad \lceil n \rceil = n$$

y para cualquier número real  $x > 0$  y enteros  $a, b > 0$ ,

$$\frac{dx=ae}{b} \quad D_I \quad \frac{x}{\text{además metro}} \quad (3.4)$$

$$\frac{bx=ac}{b} \quad D_J \quad \frac{x}{ab} \quad k; \quad (3.5)$$

$$\frac{a}{abm} \quad \frac{a \cdot b}{1/b} \quad ; \quad (3.6)$$

$$\frac{a}{abk} \quad \frac{ba \cdot b}{1/b} \quad ; \quad (3.7)$$

La función de suelo  $f(x) = \lfloor x \rfloor$  es monótonamente creciente, al igual que la función de techo  $f(x) = \lceil x \rceil$ .

### Aritmética modular

Para cualquier entero  $a$  y cualquier entero positivo  $n$ , el valor  $a \bmod n$  es el resto (o residuo) del cociente  $a=n$ :

$$a \bmod n = a - nq \quad (3.8)$$

Resulta que

$$a \equiv a \bmod n \quad (3.9)$$

Dada una noción bien definida del resto de un entero cuando se divide por otro, es conveniente proporcionar una notación especial para indicar la igualdad de los restos. Si  $a \equiv b \pmod{n}$ , escribimos  $a \equiv b \pmod{n}$  y decimos que  $a$  es equivalente a  $b$ , módulo  $n$ . En otras palabras,  $a \equiv b \pmod{n}$  si  $a$  y  $b$  tienen el mismo resto  $r$  cuando se dividen por  $n$ . De manera equivalente,  $a \equiv b \pmod{n}$  si  $n$  es un divisor de  $b-a$ . Escribimos  $a \equiv b \pmod{n}$  si  $a$  no es equivalente a  $b$ , módulo  $n$ .

### polinomios

Dado un entero no negativo d, un polinomio en n de grado d es una función  $p_n$  de la forma

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_d x^d$$

donde las constantes  $a_0, a_1, \dots, a_d$  son los coeficientes del polinomio y  $a_d \neq 0$ . Un polinomio es asintóticamente positivo si y sólo si  $a_d > 0$ . Para un polinomio asintóticamente positivo  $p_n$  de grado d, tenemos  $p_n(x) \rightarrow \infty$  para  $x \rightarrow \infty$ . Para cualquier constante real  $a < 0$ , la función  $na$  es monótonamente creciente, y para cualquier constante real  $a > 0$ , la función  $na$  es monótonamente decreciente. Decimos que una función  $f$  es polinomialmente acotada si  $|f(x)| \leq C|x|^k$  para alguna constante  $k$ .

### exponentiales

Para todos los reales  $a > 0$ ,  $m$  y  $n$ , tenemos las siguientes identidades:

$$\begin{aligned} a^0 &= 1 \\ a^1 &= a \\ a^m &= a^m \\ a^n &= a^n \\ a^m a^n &= a^{m+n} \\ a^m/a^n &= a^{m-n} \\ (a^m)^n &= a^{mn} \end{aligned}$$

Para todo  $n$  y  $a > 1$ , la función  $a^n$  es monótonamente creciente en  $n$ . Cuando sea conveniente, supondremos  $0 < a < 1$ .

Podemos relacionar las tasas de crecimiento de polinomios y exponentiales por la siguiente hecho de bajada. Para todas las constantes reales  $a$  y  $b$  tales que  $a > 1$ ,

$$\lim_{n \rightarrow \infty} \frac{a^n}{b^n} = \lim_{n \rightarrow \infty} \left(\frac{a}{b}\right)^n = \infty \quad (3.10)$$

de lo cual podemos concluir que

$$a^n > b^n \quad \text{para } n \text{ suficientemente grande}$$

Por lo tanto, cualquier función exponencial con una base estrictamente mayor que 1 crece más rápido que cualquier función polinomial.

Usando  $e$  para denotar  $2.71828 \dots$ , la base de la función logaritmo natural, tenemos para todo  $x$  real,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} \quad (3.11)$$

donde “ $\tilde{S}$ ” denota la función factorial definida más adelante en esta sección. Para todo  $x$  real, tenemos la desigualdad

$$ex \leq 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots \quad (3.12)$$

donde la igualdad se cumple solo cuando  $x = 0$ . Cuando  $|x| > 1$ , tenemos la aproximación (3.13)

$$|x| \leq 1 + |x| + \frac{|x|^2}{2!} + \dots$$

Cuando  $x \neq 0$ , la aproximación de  $ex$  por  $1 + x + \frac{x^2}{2!} + \dots$  es bastante buena:  $ex \approx 1 + x + \frac{x^2}{2!}$

$$C_x \approx 1 + x + \frac{x^2}{2!}$$

(En esta ecuación, la notación asintótica se usa para describir el comportamiento límite como  $x \rightarrow 0$  en lugar de como  $x \rightarrow 1$ .) Tenemos para todo  $x$ ,

$$\lim_{n \rightarrow \infty} \left( 1 + \frac{x}{n} \right)^n \approx ex \quad (3.14)$$

### logaritmos

Usaremos las siguientes notaciones:

$$\lg n \equiv \log_2 n \quad (\text{logaritmo binario}), \quad \ln n \equiv \log_e n$$

$$(\text{logaritmo natural}), \quad \lg_k n \equiv \lg n/k$$

$$(\text{exponenciación}), \quad \lg \lg n \equiv \lg \lg n$$

(composición).

Una importante convención de notación que adoptaremos es que las funciones logarítmicas se aplicarán sólo al siguiente término de la fórmula, de modo que  $\lg n + C k$  significará  $\lg n + C k$  y no  $\lg(n + C k)$ . Si mantenemos  $b > 1$  constante, entonces para  $n > 0$ , la función  $\log_b n$  es estrictamente creciente.

Para todos los reales  $a > 0$ ,  $b > 0$ ,  $c > 0$  y  $n, a \in D$

$$\log_b a = \frac{\ln a}{\ln b}$$

$$\log_c ab = \log_c a + \log_c b$$

$$n \log_b a = \log_b a^n$$

$$\log_b 1 = 0$$

$$\log_b a = \frac{\log_a a}{\log_a b}$$

$$alog_b c = \log_b c$$

$$D \log_b a = \log_b a$$

(3.15)

$$\log_b 1 = 0$$

$$\log_b a = \frac{\log_a a}{\log_a b}$$

$$alog_b c = \log_b c$$

(3.16)

donde, en cada ecuación anterior, las bases del logaritmo no son 1.

Por la ecuación (3.15), cambiar la base de un logaritmo de una constante a otra cambia el valor del logaritmo por solo un factor constante, por lo que a menudo usaremos la notación "lg n" cuando no nos interesan las constantes. factores, como en la notación O. Los informáticos encuentran que 2 es la base más natural para los logaritmos porque muchos algoritmos y estructuras de datos implican dividir un problema en dos partes.

Hay una expansión en serie simple para  $\ln.1 C x/$  cuando  $|x| < 1$ :

$$\ln.1 C x/ \underset{D}{\sim} \frac{x^2 x^3}{2} - \frac{x^4 x^5}{3} + \frac{x^6 x^7}{4} - \frac{x^8 x^9}{5} + \dots$$

También tenemos las siguientes desigualdades para  $x > 1$ :

$$\frac{x}{1-x} \leq \ln.1 C x/ \leq \frac{x}{1-x} ; \quad (3.17)$$

donde la igualdad se cumple solo para  $x = 1$ .

Decimos que una función  $f(n)$  está polilogarítmicamente acotada si  $f(n) \leq O(\lg k^n)$  para alguna constante  $k$ . Podemos relacionar el crecimiento de polinomios y polilogaritmos sustituyendo  $\lg n$  por  $n$  y  $2a$  por  $a$  en la ecuación (3.10), dando como resultado

$$\lim_{n \rightarrow 1} \frac{\lg b n}{2a/\lg n} \leq \lim_{n \rightarrow 1} D \leq \lim_{n \rightarrow 1} \frac{\lg b n}{n/A} \leq D \leq 0$$

A partir de este límite, podemos concluir que

$$\lg b n \leq o(na)$$

para cualquier constante  $a > 0$ . Por lo tanto, cualquier función polinomial positiva crece más rápido que cualquier función polilogarítmica.

### Factoriales

La notación  $n!$  (léase "n factorial") se define para números enteros  $n \geq 0$  como

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

$$n! \leq (1 \cdot n \cdot n-1 \cdot \dots \cdot 1) \leq n^n$$

Así,  $n! \leq 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ .

Un límite superior débil en la función factorial es  $n!$  términos en  $n^n$ , ya que cada uno de los  $n$  el producto factorial es como mucho  $n$ . aproximación de Stirling,

$$n! \underset{D}{\sim} p2 n^{\frac{n}{2}} e^{-n} \sqrt{2 \pi n} , \quad (3.18)$$

donde  $e$  es la base del logaritmo natural, nos da un límite superior más estricto y también un límite inferior. Como el Ejercicio 3.2-3 le pide que demuestre,  $n \leq D \leq n + \frac{1}{12n}$  ;

$$n \leq D \leq n + \frac{1}{12n} ; \quad \lg n \leq D \leq \lg(n+1)$$

$$D = \frac{n}{n+1} \lg n ; \text{ donde}$$

$$\text{la aproximación de Stirling} \quad (3.19)$$

es útil para probar la ecuación (3.19). La siguiente ecuación también es válida para todo  $n > 1$ :  $n \leq D \leq n + \frac{1}{12n}$

$$\frac{n}{n+1} \leq D \leq e^{-\frac{1}{12n}} \quad (3.20)$$

donde

$$\frac{1}{12n} \leq D - n \leq \frac{1}{12n} \quad (3.21)$$

#### iteración funcional

Usamos la notación  $f^{(i)}(n)$  para denotar la función  $f(n)$  aplicada iterativamente  $i$  veces a un valor inicial de  $n$ . Formalmente, sea  $f(n)$  una función sobre los reales. Para enteros no negativos  $i$ , definimos recursivamente

$$\begin{aligned} f^{(0)}(n) &= n \\ f^{(i+1)}(n) &= f(f^{(i)}(n)) \quad \text{si } i \geq 0 \end{aligned}$$

Por ejemplo, si  $f(n) = 2n$ , entonces  $f^{(i)}(n) = 2^i n$ .

#### La función logaritmo iterado

Usamos la notación  $\lg n$  (léase "log star of  $n$ ") para denotar el logaritmo iterado, definido como sigue. Sea  $\lg_i n$  como se define arriba, con  $f(n) = \lg n$ . Debido a que el logaritmo de un número no positivo no está definido,  $\lg_i n$  se define solo si  $\lg_i n > 0$ .

Asegúrese de distinguir  $\lg_i n$  (la función de logaritmo aplicada  $i$  veces seguidas, comenzando con el argumento  $n$ ) de  $\lg_i n$  (el logaritmo de  $n$  elevado a la  $i$ -ésima potencia).

Luego definimos la función de logaritmo iterado como  $0 \leq \lg_i n \leq \lg n$

$$\lg_{i+1} n = \lg(\lg_i n)$$

logaritmo iterado es una función de crecimiento muy lento:

$$\lg_2 2 = 1$$

$$\lg_4 4 = 2$$

$$\lg_{16} 16 = 3$$

$$\lg_{65536} 65536 = 4$$

$$\lg_{265536} 265536 = 5$$

Dado que el número de átomos en el universo observable se estima en alrededor de 1080, que es mucho menor que 265536, rara vez encontramos un tamaño de entrada n tal que  $\lg n > 5$ .

números de Fibonacci

Definimos los números de Fibonacci por la siguiente recurrencia:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \text{ para } i \geq 2 \end{aligned} \quad (3.22)$$

Así, cada número de Fibonacci es la suma de los dos anteriores, dando el secuencia

0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; ::::

Los números de Fibonacci están relacionados con la proporción áurea y con su conjugada y, que son las dos raíces de la ecuación.

$$\varphi = \frac{1 + \sqrt{5}}{2} \quad (3.23)$$

y están dadas por las siguientes fórmulas (ver Ejercicio 3.2-6):

$$\begin{aligned} \varphi &= \frac{1 + \sqrt{5}}{2} \\ \varphi &= \frac{1 - \sqrt{5}}{2} \\ \varphi &= \frac{\sqrt{5} - 1}{2} \end{aligned} \quad (3.24)$$

Específicamente, tenemos

$$F_i = \frac{\varphi^i - \psi^i}{\sqrt{5}}$$

que podemos probar por inducción (Ejercicio 3.2-7). Dado que  $|\psi| < 1$ , tenemos

$$\begin{aligned} \frac{\varphi^i - \psi^i}{\sqrt{5}} &< \frac{1}{\sqrt{5}} \\ &< \frac{1}{2} \end{aligned}$$

lo que implica que

$$\text{FiD} = \frac{\frac{i}{p5} C}{2} : \quad (3.25)$$

que quiere decir que el  $i$ -ésimo número de Fibonacci  $\text{Fi}$  es igual al entero  $\frac{i}{p5}$  redondeado a lo más cercano. Por lo tanto, los números de Fibonacci crecen exponencialmente.

### Ejercicios

3.2-1

Muestre que si  $f .n/$  y  $g .n/$  son funciones monótonamente crecientes, entonces también lo son las funciones  $f .n/ C g .n/$  y  $f .n// g .n//$ , y si  $f .n/$  y  $g .n/$  son además no negativas, entonces  $f .n/ g .n/$  es monótonamente creciente.

3.2-2

Demostrar la ecuación (3.16).

3.2-3

Demostrar la ecuación (3.19). Demuestre también que  $n \leq D !.2n/$  y  $n \leq D o.nn/$ .

3.2-4 ?

¿La función  $\text{dlg } n$  está polinomialmente acotada? ¿La función  $\text{lg } n$  está polinomialmente acotada?

3.2-5 ?

¿Cuál es asintóticamente mayor:  $\text{lg.lg } n$  o  $\text{lg.lg } n/$ ?

3.2-6

Muestre que la proporción áurea y su conjugado  $x_2 \leq x_1$ . ambos satisfacen la ecuación

3.2-7

Demuestre por inducción que el  $i$ -ésimo número de Fibonacci satisface la igualdad

$$\text{FiD} = \frac{\frac{i}{p5} y_0}{2} :$$

donde esta la proporción aurea y  $y_0$  es su conjugado.

3.2-8

Demuestre que  $k \ln k \leq D ,n/$  implica  $k D ,n= \ln n/$ .

**Problemas****3-1 Comportamiento asintótico de polinomios**

Dejar

d

$$pn / D X \text{ aini} ;$$

$\underset{i \geq 0}{}$

donde  $ad > 0$ , sea un polinomio de grado d en n, y sea k una constante. Use las definiciones de las notaciones asintóticas para probar las siguientes propiedades.

- a. Si  $k d$ , entonces  $pn / D O.nk/$ .
- b. Si  $k d$ , entonces  $pn / D .nk/$ .
- C. Si  $k D d$ , entonces  $pn / D ,nk/$ .
- d. Si  $k > d$ , entonces  $pn / D o.nk/$ .
- mi. Si  $k < d$ , entonces  $pn / D !.nk/$ .

**3-2 Crecimientos asintóticos**

relativos Indicar, para cada par de expresiones A; B/ en la siguiente tabla, si

A es O, o, , ! o , de B. Suponga que  $k > 0$  y  $c > 1$  son constantes. Tu respuesta debe tener la forma de la tabla con "sí" o "no" escrito en cada casilla.

A	Abucheo			!	,
a. $\lg k n$	norla				
b. $n^k$	$c^n$				
C. $p_n$	$n^{2n+2}$				
d. $2^n$					
mi. $\lg c$	$c^n$				
F. $\lg n^{\frac{1}{n}} / \lg n^n /$					

**3-3 Ordenar por tasas de crecimiento**

asintóticas a. Clasifica las siguientes funciones por orden de crecimiento; es decir, encuentre un arreglo  $g_1; g_2; \dots; g_{30}$  de las funciones que satisfacen  $g_1 D .g_2/$ ,  $g_2 D .g_3/$ , ...,  $g_{29} D .g_{30}/$ . Divide tu lista en clases de equivalencia tales que las funciones  $f .n/$  y  $g_n/$  estén en la misma clase si y solo si  $f .n/ D ,g_n//$ .

$$\begin{array}{llllll}
 \lg \lg n / & 2l \text{ norte} & . p\bar{2}/\lg & n2 & n\check{S} & .\lg n/\check{S} \\
 \cdot \frac{3}{2} \text{ norte} & n3 & n \lg 2 n \lg n \check{S} / & 22n & n1 = \lg n & \\
 \text{en } \ln n & \lg n & n 2n \lg \lg n & \text{en } n & 1 & \\
 2l \text{ norte} & .\lg n/\lg & es & 4\ln & .n C 1/\check{S} \text{ enchufe } n & \\
 n \lg \lg n / 2p2 \lg n & \underline{\underline{\text{note}}} & 2n & n \lg n & 22nC1 &
 \end{array}$$

- b. Dé un ejemplo de una sola función no negativa  $f(n)$  tal que para todas las funciones  $g(n)$  en la parte (a),  $f(n) \neq O(g(n))$  ni  $\Omega(g(n))$ .

### 3-4 Propiedades de la notación

asintótica Sean  $f(n)$  y  $g(n)$  funciones asintóticamente positivas. Demuestra o refuta cada una de las siguientes conjeturas.

- a.  $f(n) \in O(g(n))$  implica  $g(n) \in O(f(n))$ .
- b.  $f(n) \in C(g(n))$   $\Rightarrow \min(f(n); g(n))$ .
- c.  $f(n) \in O(g(n))$  implica  $\lg f(n) \in O(\lg g(n))$ , donde  $\lg g(n) \geq 1$  para todo  $n$  suficientemente grande.
- d.  $f(n) \in O(g(n))$  implica  $2f(n) \in O(2g(n))$ .
- e.  $f(n) \in O(g(n))$  implica  $f(n) \in O(g(n)/2)$ .
- f.  $f(n) \in O(g(n))$  implica  $g(n) \in O(f(n))$ .
- gramo.  $f(n) \in O(g(n))$   $\Rightarrow f(n)=2$ .
- h.  $f(n) \in C(g(n))$   $\Rightarrow f(n) \in O(g(n))$ .

### 3-5 Variaciones sobre $O$ y $\Omega$

Algunos autores definen de una manera ligeramente diferente a la nuestra; usemos  $\Omega^1$  (leer “omega infinito”) para esta definición alternativa. Decimos que  $f(n) \in \Omega^1(g(n))$  si existe una constante positiva  $c$  tal que  $f(n) \geq cg(n)$  para infinitos enteros  $n$ .

- a. Muestre que para cualesquier dos funciones  $f(n)$  y  $g(n)$  que son asintóticamente no triviales, ya sea  $f(n) \in O(g(n))$  o  $f(n) \in \Omega(g(n))$  negativas  $\Omega(g(n))$  o ambas, mientras que esto no es si usamos en lugar de  $\Omega^1$ .

- b. Describir las posibles ventajas y desventajas de utilizar caracterizar los  $^1$  en lugar de a tiempos de ejecución de los programas.

Algunos autores también definen O de una manera ligeramente diferente; usemos O0 para la definición alternativa. Decimos que  $f .n/ \leq O0 .gn//$  si y sólo si  $f .n/ \leq Ogn//$ .

- C. ¿Qué sucede con cada dirección del “si y solo si” en el Teorema 3.1 si sustituimos O0 por O pero aún usamos ?

Algunos autores definen Oe (léase “soft-oh”) en el sentido de O con factores logarítmicos ignorados:

$Ogn// \leq f ff .n/ \leq W$  existen constantes positivas c, k y  $n_0$  tales que  $0 \leq f .n/ \leq cg.n/\lgk.n/$  para todo  $n \geq n_0$  :

- d. Definir  $y ,e$  de manera similar. Demostrar el análogo correspondiente a Theorem 3.1.

### 3-6 Funciones iteradas

Podemos aplicar el operador de iteración usado en la función lg a cualquier función  $f .n/$  monótonamente creciente sobre los reales. Para una constante dada  $c \in R$ , definimos la función iterada  $f$  por

$$F_c .n/ = \min_{i=0}^n f .i/n/$$

que no es necesario definir bien en todos los casos. En otras palabras, la cantidad  $f .n/$  es el número de aplicaciones iteradas de la función  $f$  requeridas para reducir su argumento a  $c$  o menos.

Para cada una de las siguientes funciones  $f .n/$  y constantes  $c$ , dé un límite tan estrecho como sea posible en  $F_c .n/$  como  $.n/$ .

$f .n/$	$c$	$F_c .n/$
a. $n = 1$	0	
b. $\lg n$	1	
c. $n=2$ re.	1	
$n=2$ e. $p_n f .n/$	2	
$p_n = 3$	2	
—	1	
—	2	
gramo.	2	
H. $n = \text{largo norte}$	2	

### Notas del capítulo

Knuth [209] remonta el origen de la notación O a un texto de teoría de números de P. Bachmann en 1892. La notación o fue inventada por E. Landau en 1909 para su discusión sobre la distribución de los números primos. Las notaciones y , fueron defendidas por Knuth [213] para corregir la práctica popular, pero técnicamente descuidada, en la literatura de usar la notación O para los límites superior e inferior. Mucha gente continúa usando la notación O donde la notación , es técnicamente más precisa. En los trabajos de Knuth [209, 213] y Brassard y Bratley [54] aparecen más discusiones sobre la historia y el desarrollo de las notaciones asintóticas.

No todos los autores definen las notaciones asintóticas de la misma manera, aunque las distintas definiciones concuerdan en la mayoría de las situaciones comunes. Algunas de las definiciones alternativas abarcan funciones que no son asintóticamente no negativas, siempre que sus valores absolutos estén adecuadamente acotados.

La ecuación (3.20) se debe a Robbins [297]. Otras propiedades de las funciones matemáticas elementales se pueden encontrar en cualquier buena referencia matemática, como Abramowitz y Stegun [1] o Zwillinger [362], o en un libro de cálculo, como Apostol [18] o Thomas et al. [334]. Knuth [209] y Graham, Knuth y Patashnik [152] contienen una gran cantidad de material sobre matemáticas discretas tal como se utilizan en informática.

---

## 4 Divide y conquistarás

En la Sección 2.3.1, vimos cómo la ordenación por combinación sirve como ejemplo del paradigma divide y vencerás. Recuerde que en divide y vencerás, resolvemos un problema recursivamente, aplicando tres pasos en cada nivel de la recursividad:

Dividir el problema en varios subproblemas que son instancias más pequeñas del mismo problema.

Conquistar los subproblemas resolviéndolos recursivamente. Sin embargo, si los tamaños de los subproblemas son lo suficientemente pequeños, simplemente resuelva los subproblemas de una manera directa.

Combinar las soluciones de los subproblemas en la solución del problema original  
lema

Cuando los subproblemas son lo suficientemente grandes como para resolverlos recursivamente, lo llamamos el caso recursivo. Una vez que los subproblemas se vuelven lo suficientemente pequeños como para que ya no recurramos, decimos que la recursión "toca fondo" y que hemos llegado al caso base. A veces, además de los subproblemas que son instancias más pequeñas del mismo problema, tenemos que resolver subproblemas que no son exactamente iguales al problema original. Consideraremos resolver tales subproblemas como parte del paso de combinación.

En este capítulo, veremos más algoritmos basados en divide y vencerás. El primero resuelve el problema del subarreglo máximo: toma como entrada un arreglo de números y determina el subarreglo contiguo cuyos valores tienen la suma mayor.

Luego veremos dos algoritmos de divide y vencerás para multiplicar  $n \times n$  matrices. Uno se ejecuta en tiempo  $\Theta(n^3)$ , que no es mejor que el método sencillo de multiplicar matrices cuadradas. Pero el otro, el algoritmo de Strassen, se ejecuta en tiempo  $\Theta(n^{2.81})$ , lo que supera asintóticamente al método directo.

### recurrencias

Las recurrencias van de la mano con el paradigma de divide y vencerás, porque nos brindan una forma natural de caracterizar los tiempos de ejecución de los algoritmos de divide y vencerás. Una recurrencia es una ecuación o desigualdad que describe una función en términos

de su valor en entradas más pequeñas. Por ejemplo, en la Sección 2.3.2 describimos el tiempo de ejecución del peor de los casos T .n/ del procedimiento MERGE-SORT por la recurrencia

$$\begin{aligned} & \text{si } n \leq 1 \\ & T(n) = 2T(n/2) + C, \text{ si } n > 1; \end{aligned} \quad (4.1)$$

cuya solución afirmamos que es  $T(n) = C \lg n$ .

Las recurrencias pueden tomar muchas formas. Por ejemplo, un algoritmo recursivo podría dividir los subproblemas en tamaños desiguales, como una división de 2=3 a 1=3. Si los pasos de dividir y combinar toman un tiempo lineal, tal algoritmo daría lugar a la recurrencia  $T(n) = DT(n/3) + CT(n/3)$ .

Los subproblemas no están necesariamente restringidos a ser una fracción constante del tamaño del problema original. Por ejemplo, una versión recursiva de búsqueda lineal (vea el ejercicio 2.1-3) crearía solo un subproblema que contiene solo un elemento menos que el problema original. Cada llamada recursiva tomaría un tiempo constante más el tiempo de las llamadas recursivas que realiza, dando como resultado la recurrencia  $T(n) = T(n-1) + C$ .

Este capítulo ofrece tres métodos para resolver recurrencias, es decir, para obtener límites asintóticos “,” u “O” en la solución:

En el método de sustitución, adivinamos un límite y luego usamos la inducción matemática para probar que nuestra suposición es correcta.

El método del árbol de recurrencia convierte la recurrencia en un árbol cuyos nodos representan los costos incurridos en varios niveles de la recurrencia. Usamos técnicas para acotar sumatorios para resolver la recurrencia.

El método maestro proporciona límites para las recurrencias de la forma

$$T(n) = aT(n/b) + f(n); \quad (4.2)$$

donde  $a \geq 1$ ,  $b > 1$  y  $f(n)$  es una función dada. Tales recurrencias surgen con frecuencia. Una recurrencia de la forma en la ecuación (4.2) caracteriza un algoritmo divide y vencerás que crea subproblemas, cada uno de los cuales es  $1/b$  del tamaño del problema original, y en el que los pasos de dividir y combinar juntos toman  $f(n)$  tiempo.

Para usar el método maestro, deberá memorizar tres casos, pero una vez que lo haga, podrá determinar fácilmente los límites asintóticos para muchas recurrencias simples. Usaremos el método maestro para determinar los tiempos de ejecución de los algoritmos de divide y vencerás para el problema del subarreglo máximo y para la multiplicación de matrices, así como para otros algoritmos basados en divide y vencerás en otras partes de este libro.

Ocasionalmente, veremos recurrencias que no son igualdades sino desigualdades, como  $T .n/ 2T .n=2/ C ,n/$ . Debido a que tal recurrencia establece solo un límite superior en  $T .n/$ , expresaremos su solución usando notación O en lugar de notación .. De manera similar, si la desigualdad se invirtiera a  $T .n/ 2T .n=2/ C ,n/$ , entonces debido a que la recurrencia da solo un límite inferior en  $T .n/$ , usaríamos -notación en su solución.

### Tecnicismos en recurrencias

En la práctica, descuidamos ciertos detalles técnicos cuando planteamos y resolvemos recurrencias. Por ejemplo, si llamamos MERGE-SORT en n elementos cuando n es impar, terminamos con subproblemas de tamaño  $bn=2c$  y  $dn=2e$ . Ningún tamaño es en realidad  $n=2$ , porque  $n=2$  no es un número entero cuando n es impar. Técnicamente, la recurrencia que describe el peor tiempo de ejecución de MERGE-SORT es realmente

$$T .n/ D ( ,1/ T .dn=2e/ CT .bn=2c/ C ,n/ \text{ si } n>1: \text{ si } n D 1 : ) \quad (4.3)$$

Las condiciones de contorno representan otra clase de detalles que normalmente ignoramos. Dado que el tiempo de ejecución de un algoritmo en una entrada de tamaño constante es una constante, las recurrencias que surgen de los tiempos de ejecución de los algoritmos generalmente tienen  $T .n/ D ,1/$  para n suficientemente pequeño. En consecuencia, por conveniencia, generalmente omitiremos las declaraciones de las condiciones de frontera de las recurrencias y supondremos que  $T .n/$  es constante para n pequeño. Por ejemplo, normalmente declaramos recurrencia (4.1) como

$$T .n/ D 2T .n=2/ C ,n/ ; \quad (4.4)$$

sin dar explícitamente valores para n pequeño. La razón es que, aunque cambiar el valor de  $T .1/$  cambia la solución exacta de la recurrencia, la solución normalmente no cambia en más de un factor constante, por lo que el orden de crecimiento no cambia.

Cuando establecemos y resolvemos recurrencias, a menudo omitimos pisos, techos y condiciones de contorno. Seguimos adelante sin estos detalles y luego determinamos si importan o no. Por lo general, no lo hacen, pero debe saber cuándo lo hacen. La experiencia ayuda, al igual que algunos teoremas que establecen que estos detalles no afectan los límites asintóticos de muchas recurrencias que caracterizan los algoritmos de divide y vencerás (ver Teorema 4.1). En este capítulo, sin embargo, abordaremos algunos de estos detalles e ilustraremos los detalles de los métodos de solución de recurrencia.

#### 4.1 El problema del subarreglo máximo

Suponga que le ofrecen la oportunidad de invertir en Volatile Chemical Corporation. Al igual que los productos químicos que produce la empresa, el precio de las acciones de Volatile Chemical Corporation es bastante volátil. Se le permite comprar una unidad de acciones solo una vez y luego venderla en una fecha posterior, comprando y vendiendo después del cierre de operaciones del día. Para compensar esta restricción, se le permite saber cuál será el precio de las acciones en el futuro. Su objetivo es maximizar su beneficio. La figura 4.1 muestra el precio de la acción durante un período de 17 días. Puede comprar las acciones en cualquier momento, a partir del día 0, cuando el precio sea de \$100 por acción. Por supuesto, querría "comprar barato, vender caro" (comprar al precio más bajo posible y luego vender al precio más alto posible) para maximizar sus ganancias. Desafortunadamente, es posible que no pueda comprar al precio más bajo y luego vender al precio más alto dentro de un período determinado. En la Figura 4.1, el precio más bajo ocurre después del día 7, que ocurre después del precio más alto, después del día 1.

Puede pensar que siempre puede maximizar las ganancias comprando al precio más bajo o vendiendo al precio más alto. Por ejemplo, en la Figura 4.1, maximizaríamos las ganancias comprando al precio más bajo, después del día 7. Si esta estrategia siempre funcionara, entonces sería fácil determinar cómo maximizar las ganancias: encuentre los precios más altos y más bajos, y luego trabaje a la izquierda del precio más alto para encontrar el precio anterior más bajo, trabaje a la derecha desde el precio más bajo para encontrar el precio posterior más alto y tome el par con la mayor diferencia. La figura 4.2 muestra un contrajemplo simple,

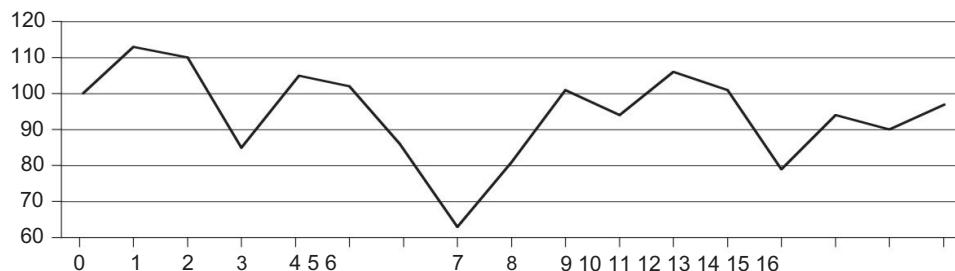


Figura 4.1 Información sobre el precio de las acciones de Volatile Chemical Corporation después del cierre de la negociación durante un período de 17 días. El eje horizontal del gráfico indica el día y el eje vertical muestra el precio. La fila inferior de la tabla muestra el cambio en el precio del día anterior.

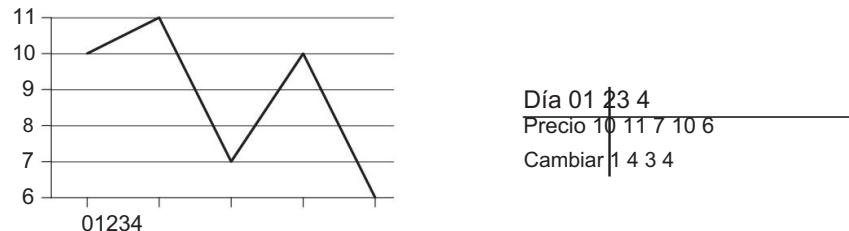


Figura 4.2 Un ejemplo que muestra que la ganancia máxima no siempre comienza con el precio más bajo o termina con el precio más alto. Nuevamente, el eje horizontal indica el día y el eje vertical muestra el precio. Aquí, la ganancia máxima de \$3 por acción se obtendría comprando después del día 2 y vendiendo después del día 3. El precio de \$7 después del día 2 no es el precio más bajo en general, y el precio de \$10 después del día 3 no es el precio más alto. en general.

demostrando que el máximo beneficio a veces no se obtiene ni comprando al precio más bajo ni vendiendo al precio más alto.

#### Una solución de fuerza bruta

Podemos idear fácilmente una solución de fuerza bruta para este problema: simplemente pruebe todos los pares posibles de fechas de compra y venta en las que la fecha de compra preceda a la fecha de venta. Un período de  $n$  es  $\binom{n}{2}$ , y lo mejor que podemos esperar de días tiene tales pares de  $\binom{n}{2}$  venta. Como se trata de evaluar cada par de fechas en tiempo constante, este enfoque tomaría  $\binom{n}{2}$  tiempo. ¿Podemos hacerlo mejor?

#### Una transformación

Para diseñar un algoritmo con un tiempo de ejecución  $O(n^2)$ , veremos la entrada de una manera ligeramente diferente. Queremos encontrar una secuencia de días en la que el cambio neto desde el primer día hasta el último sea máximo. En lugar de mirar los precios diarios, consideraremos el cambio diario en el precio, donde el cambio en el día  $i$  es la diferencia entre los precios después del día  $i$  y después del día  $i$ . La tabla de la Figura 4.1 muestra estos cambios diarios en la fila inferior. Si tratamos esta fila como un arreglo  $A$ , como se muestra en la figura 4.3, ahora queremos encontrar el subarreglo contiguo no vacío de  $A$  cuyos valores tienen la suma más grande. Llamamos a este subarreglo contiguo el subarreglo máximo. Por ejemplo, en el arreglo de la Figura 4.3, el subarreglo máximo de  $A[1 : : 16]$  es  $A[8 : : 11]$ , con la suma 43. Por lo tanto, querrá comprar las acciones justo antes del día 8 (es decir, después del día 7) y venderlo después del día 11, obteniendo una ganancia de \$43 por acción.

A primera vista, esta transformación no ayuda. Todavía necesitamos verificar los subarreglos  $D_{n1}^{n2}$  por un período de  $n$  días. El ejercicio 4.1-2 le pide que muestre

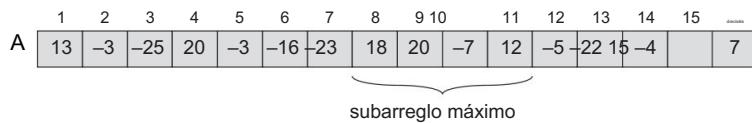


Figura 4.3 El cambio en los precios de las acciones como un problema de subarreglo máximo. Aquí, el subarreglo  $A[8 : : 11]$ , con suma 43, tiene la mayor suma de cualquier subarreglo contiguo del arreglo A.

que aunque calcular el costo de un subarreglo puede tomar un tiempo proporcional a la longitud del subarreglo, al calcular todas las sumas de los subarreglos  $.n^2/$ , podemos organizar el cómputo de manera que cada suma de los subarreglos tome  $0.1/n^2$  tiempo, dados los valores de sumas de subarreglo previamente calculadas, de modo que la solución de fuerza bruta toma  $.n^2/$  tiempo.

Entonces, busquemos una solución más eficiente para el problema del subarreglo máximo. Al hacerlo, normalmente hablaremos de "un" subarreglo máximo en lugar de "el" subarreglo máximo, ya que podría haber más de un subarreglo que alcance la suma máxima.

El problema del subarreglo máximo es interesante solo cuando el arreglo contiene algunos números negativos. Si todas las entradas del arreglo fueran no negativas, entonces el problema del subarreglo máximo no presentaría ningún desafío, ya que el arreglo completo daría la suma mayor.

#### Una solución usando divide y vencerás

Pensemos en cómo podríamos resolver el problema del subarreglo máximo usando la técnica divide y vencerás. Supongamos que queremos encontrar un rayo de subarreglo máximo del subarreglo  $A[low : : high]$ . Divide y vencerás sugiere que dividimos el subarreglo en dos subarreglos del mismo tamaño posible. Es decir, encontramos el punto medio, digamos  $mid$ , del subarreglo, y consideramos los subarreglos  $A[low : : mid]$  y  $A[mid + 1 : : high]$ . Como muestra la Figura 4.4(a), cualquier subarreglo contiguo  $A[i : : j]$  de  $A[low : : high]$  debe estar exactamente en uno de los siguientes lugares:

completamente en el subarreglo  $A[low : : mid]$ , de modo que  $low \leq i \leq j \leq mid$ ,

completamente en el subarreglo  $A[mid + 1 : : high]$ , de modo que  $mid < i \leq j \leq high$ , o

cruzando el punto medio, de modo que  $low \leq i \leq mid < j \leq high$ .

Por lo tanto, un subarreglo máximo de  $A[low : : high]$  debe estar exactamente en uno de estos lugares. De hecho, un subarreglo máximo de  $A[low : : high]$  debe tener la mayor suma sobre todos los subarreglos enteramente en  $A[low : : mid]$ , enteramente en  $A[mid + 1 : : high]$ , o cruzando el punto medio. Podemos encontrar subarreglos máximos de  $A[low : : mid]$  y  $A[mid + 1 : : high]$  recursivamente, porque estos dos subproblemas son instancias más pequeñas del problema de encontrar un subarreglo máximo. Por lo tanto, todo lo que queda por hacer es encontrar un

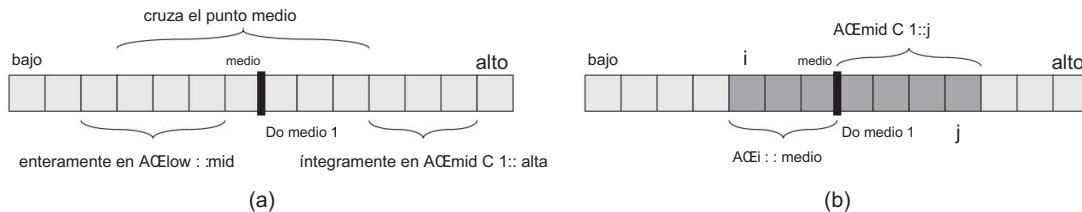


Figura 4.4 (a) Posibles ubicaciones de los subarreglos de  $A[low : : high]$ : enteramente en  $A[low : : mid]$ , enteramente en  $A[mid : : high]$ , o cruzando el punto medio  $mid$ . (b) Cualquier subarreglo de  $A[low : : high]$  que cruce el punto medio comprende dos subarreglos  $A[i : : mid]$  y  $A[mid + 1 : : j]$ , donde  $low \leq i < mid < j \leq high$ .

subarreglo mximo que cruza el punto medio, y tome un subarreglo con la suma ms grande de los tres.

Podemos encontrar fácilmente un subarreglo máximo que cruce el punto medio en el tiempo lineal en el tamaño del subarreglo  $A[low : : high]$ . Este problema no es una instancia más pequeña de nuestro problema original, porque tiene la restricción adicional de que el subarreglo que elija debe cruzar el punto medio. Como muestra la Figura 4.4(b), cualquier subarreglo que cruce el punto medio está hecho de dos subarreglos  $A[i : : mid]$  y  $A[mid + 1 : : j]$ , donde  $low \leq i < mid$  y  $mid < j \leq high$ . Por lo tanto, solo necesitamos encontrar subarreglos máximos de la forma  $A[i : : mid]$  y  $A[mid + 1 : : j]$  y luego combinarlos. El procedimiento FIND-MAX-CROSSING-SUBARRAY toma como entrada el arreglo  $A$  y los índices bajo, medio y alto, y devuelve una tupla que contiene los índices que delimitan un subarreglo máximo que cruza el punto medio, junto con la suma de los valores en un subarreglo máximo.

ENCONTRAR-MAX-CRUCE-SUBARRRAY.A; bajo; medio; alto/

```

1 suma izquierda D 1
2 suma D 0
3 para i D de medio a bajo
4         sum D sum C AŒi if
5         sum > left-sum left-
6         sum D sum max-
7         left D i 8 right-
sum D 1 9 sum D 0 10
for j D mid C 1

to high sum D sum C AŒj 11 12 if
        sum > suma-derecha
13 suma-derecha D suma 14
max-derecha D j 15 return .max-
izquierda; max-derecha; suma-
izquierda C suma-derecha/

```

Este procedimiento funciona de la siguiente manera. Las líneas 1 a 7 encuentran un subarreglo máximo de la mitad izquierda,  $A[i : : mid]$ . Dado que este subarreglo debe contener  $A[mid : : mid]$ , el ciclo for de las líneas 3–7 inicia el índice  $i$  en el medio y baja hasta el bajo, de modo que cada subarreglo que considera tiene la forma  $A[i : : mid]$ . Las líneas 1 y 2 inicializan las variables  $left-sum$ , que contiene la mayor suma encontrada hasta ahora, y  $sum$ , que contiene la suma de las entradas en  $A[i : : mid]$ . Siempre que encontremos, en la línea 5, un medio con una suma de subarreglo  $A[i : : mid]$  valores mayores que  $left-sum$ , actualizamos  $left-sum$  a la suma de este subarreglo en la línea 6, y en la línea 7 actualizamos la variable  $max-left$  para registrar este índice  $i$ . Las líneas 8–14 funcionan de manera análoga para la mitad derecha,  $A[mid : : C1:: high]$ . Aquí, el bucle for de las líneas 10–14 inicia el índice  $j$  en  $mid+1$  y trabaja hasta alto, de modo que cada subarreglo que considera tiene la forma  $A[mid : : C1:: j]$ . Finalmente, la línea 15 devuelve los índices  $max-left$  y  $max-right$  que delimitan un subarreglo máximo que cruza el punto medio, junto con la suma  $left-sum+right-sum$  de los valores en el subarreglo  $A[max-left : : max-right]$ .

Si el subarreglo  $A[i : : high]$  contiene  $n$  entradas (de modo que  $n \geq high - i + 1$ ), afirmamos que la llamada `FIND-MAX-CROSSING-SUBARRAY(A; bajo; medio; alto)` toma  $\frac{n}{2}$  tiempo. Dado que cada iteración de cada uno de los dos ciclos for toma  $\frac{1}{2}$  tiempo, solo necesitamos contar cuántas iteraciones hay en total. El bucle for de las líneas 3 a 7 hace iteraciones de  $C_1$  medio-bajo, y el bucle for de las líneas 10–14 hace iteraciones de medio alto, por lo que el número total de iteraciones es

$\dots \cdot \text{medio bajo } C_1 / C \cdot \text{alto medio} / D \cdot \text{alto bajo } C_1 \dots$

$D_n$

Con un procedimiento `FIND-MAX-CROSSING-SUBARRAY` de tiempo lineal en la mano, podemos escribir un pseudocódigo para un algoritmo de divide y vencerás para resolver el problema del subarreglo máximo:

```

ENCONTRAR-MAXIMO-SUBARRAY(A; bajo; alto)
    1 si alto == bajo 2
    devuelve .bajo; alto; A[i : : mid] // caso base: solo un elemento
    D b.low C high/=2c 4 .left-low; izquierda-
    alto; suma-izquierda/ D ENCONTRAR-
        SUBARRAY-MÁXIMO(A; bajo; mid / .right-low; right-
    5     high; right-sum/ D FIND-MAXIMUM-
        SUBRAY.A; do medio 1; alto / .cruzado-bajo; alto cruzado;
    6     cross-sum/ D FIND-MAX-CROSSING-
        SUBRAY.A; bajo; medio; alto/
    7     if left-sum right-sum y left-sum cross-sum devuelven .left-
    8         low; izquierda-alto; left-sum/ elseif right-
    9         sum left-sum y right-sum cross-sum return .right-low; right-high; right-
    10        sum/ else return .cross-low; alto cruzado; suma
    11        cruzada/

```

La llamada inicial FIND-MAXIMUM-SUBARRAY.A; 1; A:longitud/ encontrará un subarreglo máximo de AŒ1: Similar a FIND- norte.

MAX-CROSSING-SUBARRAY, el procedimiento recursivo FIND MAXIMUM-SUBARRAY devuelve una tupla que contiene los índices que delimitan un subarreglo máximo, junto con la suma de los valores en un subarreglo máximo.

La línea 1 prueba el caso base, donde el subarreglo tiene solo un elemento. Un rayo de subarreglo con un solo elemento tiene solo un subarreglo, en sí mismo, por lo que la línea 2 devuelve una tupla con los indices inicial y final de un solo elemento, junto con su valor. Las líneas 3 a 11 manejan el caso recursivo. La línea 3 hace la parte de la división, calculando el índice medio del punto medio. Vamos a referirnos al subarreglo AŒlow : : mid como el subarreglo izquierdo ya AŒmidC 1:: high como el subarreglo derecho. Como sabemos que el subarreglo AŒlow : : high contiene al menos dos elementos, cada uno de los subarreglos izquierdo y derecho debe tener al menos un elemento. Las líneas 4 y 5 conquistan al encontrar recursivamente los subarreglos máximos dentro de los subarreglos izquierdo y derecho, respectivamente.

Las líneas 6 a 11 forman la parte combinada. La línea 6 encuentra un subarreglo máximo que cruza el punto medio. (Recuerde que debido a que la línea 6 resuelve un subproblema que no es una instancia más pequeña del problema original, consideraremos que está en la parte combinada). La línea 7 prueba si el subarreglo izquierdo contiene un subarreglo con la suma máxima, y la línea 8 devuelve ese subarreglo máximo. De lo contrario, la línea 9 comprueba si el subarreglo de la derecha contiene un subarreglo con la suma máxima, y la línea 10 devuelve ese subarreglo máximo. Si ni el subarreglo izquierdo ni el derecho contienen un subarreglo que alcance la suma máxima, entonces un subarreglo máximo debe cruzar el punto medio y la línea 11 lo devuelve.

### Analizando el algoritmo divide y vencerás

A continuación, configuramos una recurrencia que describe el tiempo de ejecución del procedimiento recursivo FIND MAXIMUM-SUBARRAY . Como hicimos cuando analizamos la ordenación por fusión en la Sección 2.3.2, hacemos la suposición simplificadora de que el tamaño del problema original es una potencia de 2, de modo que todos los tamaños de los subproblemas son números enteros. Denotamos por T .n/ el tiempo de ejecución de FIND-MAXIMUM-SUBARRAY en un subarreglo de n elementos. Para empezar, la línea 1 toma tiempo constante. El caso base, cuando n D 1, es fácil: la línea 2 toma un tiempo constante, y así

$$T .1/ D ,.1/ : \quad (4.5)$$

El caso recursivo ocurre cuando n>1. Las líneas 1 y 3 toman tiempo constante. Cada uno de los subproblemas resueltos en las líneas 4 y 5 está en un subarreglo de n=2 elementos (nuestra suposición de que el tamaño del problema original es una potencia de 2 asegura que n=2 es un número entero), por lo que gastamos T .n= 2/ tiempo resolviendo cada uno de ellos. Debido a que tenemos que resolver dos subproblemas, para el subarreglo izquierdo y para el subarreglo derecho, la contribución al tiempo de ejecución de las líneas 4 y 5 llega a 2T .n=2/. como tenemos

Ya visto, la llamada a FIND-MAX-CROSSING-SUBARRAY en la línea 6 toma  $n/$  tiempo. Las líneas 7–11 toman solo  $1/$  tiempo. Para el caso recursivo, por lo tanto, tenemos  $T(n) = D(n) + C(n)$

$$2T(n/2) + C(n/2) \leq C(n/2)$$

$$D(2T(n/2) + C(n/2)) \quad (4.6)$$

La combinación de las ecuaciones (4.5) y (4.6) nos da una recurrencia para el tiempo de ejecución  $T(n)$  de ENCONTRAR-SUBRAYO-MÁXIMO:

$$\begin{aligned} T(n) &= D(n) + C(n) \\ &= D(n) + 2T(n/2) + C(n/2) \quad \text{si } n > 1 \\ &= C(n) \quad \text{si } n = 1 \end{aligned} \quad (4.7)$$

Esta recurrencia es la misma que la recurrencia (4.1) para la ordenación por fusión. Como veremos del método maestro en la Sección 4.5, esta recurrencia tiene la solución  $T(n) = D(n) + n \lg n$ . También puede revisar el árbol de recurrencia de la figura 2.5 para comprender por qué la solución debe ser  $T(n) = D(n) + n \lg n$ .

Por lo tanto, vemos que el método divide y vencerás produce un algoritmo que es asintóticamente más rápido que el método de fuerza bruta. Con merge sort y ahora el problema del máximo de subarreglos, comenzamos a tener una idea de cuán poderoso puede ser el método divide y vencerás. A veces producirá el algoritmo asintóticamente más rápido para un problema, y otras veces podemos hacerlo aún mejor. Como muestra el ejercicio 4.1-5, de hecho existe un algoritmo de tiempo lineal para el problema del subarreglo máximo y no usa divide y vencerás.

### Ejercicios

#### 4.1-1

¿Qué devuelve FIND-MAXIMUM-SUBARRAY cuando todos los elementos de A son negativos?

#### 4.1-2

Escriba un pseudocódigo para el método de fuerza bruta para resolver el problema del subarreglo máximo. Su procedimiento debe ejecutarse en  $n^2$  tiempo.

#### 4.1-3

Implemente los algoritmos recursivo y de fuerza bruta para el problema del subarreglo máximo en su propia computadora. ¿Qué tamaño de problema  $n_0$  da el punto de cruce en el que el algoritmo recursivo vence al algoritmo de fuerza bruta? Luego, cambie el caso base del algoritmo recursivo para usar el algoritmo de fuerza bruta siempre que el tamaño del problema sea menor que  $n_0$ . ¿Eso cambia el punto de cruce?

#### 4.1-4

Supongamos que cambiamos la definición del problema del subarreglo máximo para permitir que el resultado sea un subarreglo vacío, donde la suma de los valores de un subarreglo vacío

ray es 0. ¿Cómo cambiaría cualquiera de los algoritmos que no permiten subarreglos vacíos para permitir que el resultado sea un subarreglo vacío?

#### 4.1-5

Use las siguientes ideas para desarrollar un algoritmo de tiempo lineal no recursivo para el problema del subarreglo máximo. Comience en el extremo izquierdo de la matriz y progrese hacia la derecha, realizando un seguimiento del subarreglo máximo visto hasta el momento. Conociendo un subarreglo máximo de  $A[i:j, C[1:n]]$ , extienda la respuesta para encontrar un subarreglo máximo que termine en  $C[1:n]$  usando la siguiente observación: un subarreglo máximo de  $A[i:j, C[1:n]]$  es un subarreglo máximo de  $A[i:j, C[1:n]]$  o un subarreglo  $A[i:j, C[1:n]]$ ,  
 $i < j$  para algún  $j \leq n$ . Determine un subarreglo máximo de la forma  $A[i:j, C[1:n]]$  en tiempo constante basado en conocer un subarreglo máximo que termine en el índice  $j$ .

## 4.2 Algoritmo de Strassen para la multiplicación de matrices

Si has visto matrices antes, entonces probablemente sepas cómo multiplicarlas.

(De lo contrario, debe leer la Sección D.1 en el Apéndice D.) Si  $A$  y  $B$  son matrices cuadradas de  $n \times n$ , entonces en el producto  $CD = A \cdot B$ , definimos para  $i, j \in \{1, 2, \dots, n\}$ , por entrada  $c_{ij}$ ,

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad (4.8)$$

Debemos calcular  $n^2$  entradas de matriz, y cada una es la suma de  $n$  valores. El siguiente procedimiento toma las matrices  $A$  y  $B$  de  $n \times n$  y las multiplica, devolviendo su producto  $C$  de  $n \times n$ . Suponemos que cada matriz tiene un atributo `filas`, lo que da el número de filas en la matriz.

```

CUADRADO-MATRIZ-MULTIPLICAR(A; B;
1 n D A: filas
2 sea C una nueva matriz nn
3 para i D 1 a n
4     para j D 1 a n cij
5         D 0 para
6         k D 1 a n cij D cij
7         C aik bkj
8 volver C

```

El procedimiento `SQUARE-MATRIX-MULTIPLY` funciona de la siguiente manera. El ciclo for de las líneas 3 a 7 calcula las entradas de cada fila  $i$ , y dentro de una fila dada  $i$ , el

for loop de las líneas 4–7 calcula cada una de las entradas  $c_{ij}$ , para cada columna  $j$ . La línea 5 inicializa  $c_{ij}$  en 0 cuando comenzamos a calcular la suma dada en la ecuación (4.8), y cada iteración del ciclo for de las líneas 6 y 7 agrega un término más de la ecuación (4.8).

Debido a que cada uno de los bucles for triplemente anidados ejecuta exactamente  $n$  iteraciones, y cada ejecución de la línea 7 toma un tiempo constante, el procedimiento SQUARE-MATRIX-MULTIPLY toma  $.n^3$  tiempo.

Al principio, podría pensar que cualquier algoritmo de multiplicación de matrices debe tomar  $.n^3$  de tiempo, ya que la definición natural de la multiplicación de matrices requiere tantas multiplicaciones. Sin embargo, estaría equivocado: tenemos una forma de multiplicar matrices en tiempo  $O(n^2)$ . En esta sección, veremos el notable algoritmo recursivo de Strassen para multiplicar matrices de  $n \times n$ . Funciona en tiempo  $\lg 7$ , que mostraremos en la Sección 4.5. Dado que  $\lg 7$  se encuentra entre 2.80 y 2.81, el algoritmo de Strassen se ejecuta en tiempo  $O(2.81n^2)$ , que es asintóticamente mejor que el simple procedimiento SQUARE-MATRIX MULTIPLY .

#### Un simple algoritmo de divide y vencerás

Para simplificar las cosas, cuando usamos un algoritmo divide y vencerás para calcular el producto de matrices  $CDAB$ , suponemos que  $n$  es una potencia exacta de 2 en cada una de las  $n \times n$  matrices. Hacemos esta suposición porque en cada paso de división, dividiremos  $n \times n$  matrices en cuatro matrices  $n=2$   $n=2$ , y al suponer que  $n$  es una potencia exacta de 2, estamos garantizados que siempre que  $n \geq 2$ , la dimensión  $n = 2$  es un número entero.

Supongamos que dividimos cada uno de  $A$ ,  $B$  y  $C$  en cuatro matrices  $n=2$   $n=2$

$$\begin{array}{lll} \text{ANUNCIO} & A_{11} A_{12} & B_{11} B_{12} \\ & A_{21} A_{22} ; & B_{21} B_{22} \end{array} \quad \vdots \quad \begin{array}{ll} CD & C_{11} C_{12} \\ & C_{21} C_{22} ; \end{array} \quad (4.9)$$

de modo que reescribimos la ecuación  $CDAB$  como

$$\begin{array}{lll} C_{11} C_{12} & A_{11} A_{12} & B_{11} B_{12} \\ C_{21} C_{22} & A_{21} A_{22} & B_{21} B_{22} \end{array} \quad (4.10)$$

La ecuación (4.10) corresponde a las cuatro ecuaciones

$$C_{11} D A_{11} B_{11} C A_{12} B_{21} ; \quad (4.11)$$

$$C_{12} D A_{11} B_{12} C A_{12} B_{22} ; \quad (4.12)$$

$$C_{21} D A_{21} B_{11} C A_{22} B_{21} ; \quad (4.13)$$

$$C_{22} D A_{21} B_{12} C A_{22} B_{22} : \quad (4.14)$$

Cada una de estas cuatro ecuaciones especifica dos multiplicaciones de matrices  $n=2$   $n=2$  y la suma de sus productos  $n=2$   $n=2$ . Podemos usar estas ecuaciones para crear un algoritmo directo, recursivo, de divide y vencerás:

```

MATRIZ-CUADRADA-MULTIPLICAR-RECURSIVA.A; B/
1 n D A: filas 2 sea
C una nueva matriz nn 3 si n == 1 4
c11 D a11 b11
5 sino divida A , B y C como
en las ecuaciones (4.9) .A11; B11/ C MATRIZ CUADRADA-
MULTIPLICAR-RECURSIVA.A12; B21/ C12 D MATRIZ CUADRADA-MULTIPLICAR-
RECURSIVA.A11; B12/ C MATRIZ CUADRADA-MULTIPLICAR-
7 RECURSIVA.A12; B22/ C21 D MATRIZ CUADRADA-MULTIPLICAR-
RECURSIVA.A21; B11/ C MATRIZ CUADRADA-MULTIPLICAR-
8 RECURSIVA.A22; B21/ C22 D MATRIZ CUADRADA-MULTIPLICAR-
RECURSIVA.A21; B12/ C MATRIZ CUADRADA-MULTIPLICAR-
9 RECURSIVA.A22; B22/ 10 retorno C

```

Este pseudocódigo pasa por alto un detalle de implementación sutil pero importante. ¿Cómo dividimos las matrices en la línea 5? Si fuéramos a crear 12 nuevas matrices  $n=2n=2$ , gastaríamos  $.n2/$  tiempo copiando entradas. De hecho, podemos particionar las matrices sin copiar entradas. El truco es usar cálculos de índice. Identificamos una submatriz por un rango de índices de fila y un rango de índices de columna de la matriz original. Terminamos representando una submatriz de manera un poco diferente de cómo representamos la matriz original, que es la sutileza que estamos pasando por alto.

La ventaja es que, dado que podemos especificar submatrices mediante cálculos de índice, la ejecución de la línea 5 requiere solo  $.1/$  tiempo (aunque veremos que no hace ninguna diferencia asintóticamente en el tiempo de ejecución general si copiamos o particionamos en el lugar).

Ahora, derivamos una recurrencia para caracterizar el tiempo de ejecución de **MATRIZ CUADRADA -MULTIPLICAR-RECURSIVA**. Sea  $T .n/$  el tiempo para multiplicar dos matrices  $nn$  usando este procedimiento. En el caso base, cuando  $n = 1$ , realizamos solo una multiplicación escalar en la línea 4, y así

$$T .1/ \leq D .1/ : \quad (4.15)$$

El caso recursivo ocurre cuando  $n > 1$ . Como se discutió, dividir las matrices en la línea 5 toma  $.1/$  tiempo, usando cálculos de índice. En las líneas 6 a 9, recursivamente llamamos **CUADRADO-MATRIZ-MULTIPLICAR-RECURSIVO** un total de ocho veces. Debido a que cada llamada recursiva multiplica dos matrices  $n=2 n=2$ , contribuyendo así  $T .n=2/$  al tiempo total de ejecución, el tiempo que tardan las ocho llamadas recursivas es  $8T .n=2/$ . Nosotros también debe tener en cuenta las cuatro adiciones de matriz en las líneas 6 a 9. Cada una de estas matrices contiene  $n=4$  entradas, por lo que cada una de las cuatro adiciones de matriz toma  $.n2/$  tiempo. Dado que el número de adiciones de matriz es una constante, el tiempo total dedicado a agregar ma-

trices en las líneas 6 a 9 es  $,n^2/$ . (Nuevamente, usamos cálculos de índice para colocar los resultados de las adiciones de la matriz en las posiciones correctas de la matriz C, con una sobrecarga de  $,1/$  tiempo por entrada). El tiempo total para el caso recursivo, por lo tanto, es la suma de el tiempo de partición, el tiempo para todas las llamadas recursivas y el tiempo para sumar las matrices resultantes de las llamadas recursivas:

$$\begin{aligned} T .n/ D ,1/ C & 8T .n=2/ C ,n2/ D 8T .n=2/ C \\ ,n2 / : & \end{aligned} \quad (4.16)$$

Observe que si implementáramos la partición mediante la copia de matrices, lo que costaría  $,n^2/$  tiempo, la recurrencia no cambiaría y, por lo tanto, el tiempo total de ejecución aumentaría solo en un factor constante.

La combinación de las ecuaciones (4.15) y (4.16) nos da la recurrencia de la ejecución tiempo de CUADRADO-MATRIZ-MULTIPLICAR-RECURSIVO:

$$\begin{aligned} \text{si } n & D 1 \\ T .n/ D ( ,1/ 8T .n=2/ C ,n2/ \text{ si } n>1: & \end{aligned} \quad (4.17)$$

Como veremos del método maestro en la Sección 4.5, la recurrencia (4.17) tiene la solución  $T .n/ D ,n^3/$ . Por lo tanto, este enfoque simple de divide y vencerás no es más rápido que el procedimiento directo de MULTIPLICACIÓN DE MATRIZ CUADRADA .

Antes de continuar con el examen del algoritmo de Strassen, revisemos de dónde provienen los componentes de la ecuación (4.16). Dividir cada matriz  $n^n$  por cálculo de índice toma  $,1/$  tiempo, pero tenemos dos matrices para dividir. Aunque se podría decir que dividir las dos matrices toma  $,2/$  tiempo, la constante de 2 se subsume en la notación  $,$ . Sumar dos matrices, cada una con, digamos,  $k$  entradas, lleva  $,k/$  tiempo. Dado que cada una de las matrices que sumamos tiene  $n^2=4$  entradas, se podría decir que sumar cada par lleva  $,n^2=4/$  tiempo. De nuevo, sin embargo, la notación  $,$  subsume el factor constante de  $1=4$ , y decimos que sumar dos matrices  $n^2=4$   $n^2=4$  toma  $,n^2/$  tiempo. Tenemos cuatro adiciones de matriz de este tipo, y una vez más, en lugar de decir que toman  $,4n^2/$  tiempo, decimos que toman  $,n^2/$  tiempo.

(Por supuesto, podría observar que podríamos decir que las cuatro adiciones de matrices toman  $,4n^2=4/$  tiempo, y que  $4n^2=4 D n^2$ , pero el punto aquí es que la notación  $,$ -subsume factores constantes, sean cuales sean. ) Así, terminamos con dos términos de  $,n^2/$ , que podemos combinar en uno.

Sin embargo, cuando tomamos en cuenta las ocho llamadas recursivas, no podemos simplemente subsumir el factor constante de 8. En otras palabras, debemos decir que juntas toman  $8T .n=2/$  tiempo, en lugar de solo  $T .n=2/$  tiempo. Puede tener una idea de por qué mirando hacia atrás en el árbol de recurrencia en la Figura 2.5, para la recurrencia (2.1) (que es idéntica a la recurrencia (4.7)), con el caso recursivo  $T .n/ D 2T .n=2/C ,norte/$ . El factor de 2 determinaba cuántos hijos tenía cada nodo del árbol, lo que a su vez determinaba cuántos términos contribuían a la suma en cada nivel del árbol. Si tuviéramos que ignorar

el factor de 8 en la ecuación (4.16) o el factor de 2 en la recurrencia (4.1), el árbol de recurrencia sería simplemente lineal, en lugar de "tupido", y cada nivel contribuiría solo con un término a la suma.

Tenga en cuenta, por lo tanto, que aunque la notación asintótica subsume constantes factores multiplicativos, notación recursiva como  $T .n=2/$  no.

#### método de strassen

La clave del método de Strassen es hacer que el árbol de recurrencia sea un poco menos tupido. Es decir, en lugar de realizar ocho multiplicaciones recursivas de matrices  $n=2$   $n=2$ , realiza solo siete. El costo de eliminar una multiplicación de matrices será varias adiciones nuevas de matrices  $n=2$   $n=2$ , pero solo un número constante de adiciones. Como antes, el número constante de adiciones a la matriz se incluirá en la notación , cuando configuremos la ecuación de recurrencia para caracterizar el tiempo de ejecución.

El método de Strassen no es del todo obvio. (Este podría ser el mayor eufemismo en este libro.) Tiene cuatro pasos:

1. Divida las matrices de entrada A y B y la matriz de salida C en  $n=2n=2$  submatrices, como en la ecuación (4.9). Este paso toma  $.1/$  tiempo por cálculo de índice, tal como en CUADRADO-MATRIZ-MULTIPLICACIÓN-RECURSIVA.
2. Crear 10 matrices  $S_1; S_2; \dots; S_{10}$ , cada uno de los cuales es  $n=2$   $n=2$  y es la suma o diferencia de dos matrices creadas en el paso 1. Podemos crear las 10 matrices en  $.n2/$  tiempo .
3. Usando las submatrices creadas en el paso 1 y las 10 matrices creadas en el paso 2, calcule recursivamente siete productos de matriz  $P_1; P_2; \dots; P_7$ . Cada matriz  $P_i$  es  $n=2$   $n=2$ .
4. Calcular las submatrices deseadas  $C_{11}; C_{12}; C_{21}; C_{22}$  de la matriz resultante C sumando y restando varias combinaciones de las matrices  $P_i$  . Podemos calcular las cuatro submatrices en  $.n2/$  tiempo.

Veremos los detalles de los pasos 2 a 4 en un momento, pero ya tenemos suficiente información para configurar una recurrencia para el tiempo de ejecución del método de Strassen. Supongamos que una vez que el tamaño de la matriz n llega a 1, realizamos una simple multiplicación escalar, tal como en la línea 4 de SQUARE-MATRIX-MULTIPLY-RECURSIVE. Cuando  $n>1$ , los pasos 1, 2 y 4 toman un total de  $.n2/$  tiempo, y el paso 3 requiere que realicemos siete multiplicaciones de matrices  $n=2$   $n=2$ . Por lo tanto, obtenemos la siguiente recurrencia para el tiempo de ejecución T .n/ del algoritmo de Strassen:

$$T .n/ = D \left( \begin{array}{l} 1 & \text{si } n = 1 \\ 7T .n/2 + C .n2/ & \text{si } n > 1 \end{array} \right) \quad (4.18)$$

Hemos intercambiado una multiplicación de matrices por un número constante de adiciones de matrices. Una vez que entendamos las recurrencias y sus soluciones, veremos que esta compensación en realidad conduce a un tiempo de ejecución asintótico más bajo. Por el método maestro de la Sección 4.5, la recurrencia (4.18) tiene la solución  $T(n) \leq D \cdot n^2 \lg 7$ .

Ahora procedemos a describir los detalles. En el paso 2, creamos las siguientes 10 matrices:

S1 D B12 B22 ;

S2 D A11 C A12 ;

S3 D A21 C A22 ;

S4 D B21 B11 ;

S5 D A11 C A22 ;

S6 D B11 C B22 ;

S7 D A12 A22 ;

S8 D B21 C B22 ;

S9 D A11 A21 ;

S10 D B11 C B12 :

Dado que debemos sumar o restar matrices  $n=2$   $n=2$  10 veces, este paso de hecho toma  $\Theta(n^2)$  tiempo.

En el paso 3, multiplicamos recursivamente las matrices  $n=2$   $n=2$  siete veces para calcular las siguientes matrices  $n=2$   $n=2$ , cada una de las cuales es la suma o diferencia de los productos de las submatrices A y B:

P1 D A11 S1 D A11 B12 A11 B22 ;

P2 D S2 B22 D A11 B22 C A12 B22 ;

P3 D S3 B11 D A21 B11 C A22 B11 ;

P4 D A22 S4 D A22 B21 A22 B11 ;

P5 D S5 S6 D A11 B11 C A11 B22 C A22 B11 C A22 B22 ;

P6 D S7 S8 D A12 B21 C A12 B22 A22 B21 A22 B22 ;

P7 D S9 S10 D A11 B11 C A11 B12 A21 B11 A21 B12 :

Tenga en cuenta que las únicas multiplicaciones que necesitamos realizar son las que se encuentran en la columna del medio de las ecuaciones anteriores. La columna de la derecha muestra a qué equivalen estos productos en términos de las submatrices originales creadas en el paso 1.

El paso 4 suma y resta las matrices  $P_i$  creadas en el paso 3 para construir las cuatro submatrices  $n=2$   $n=2$  del producto C. Empezamos con

C11 D P5 C P4 P2 C P6 :

Expandiendo el lado derecho, con la expansión de cada  $P_i$  en su propia línea y alineando verticalmente los términos que se cancelan, vemos que  $C_{11}$  es igual a

$$\begin{array}{ccccccc}
 A_{11} & B_{11} & C A_{11} & B_{22} & C A_{22} & B_{11} & C A_{22} B_{22} \\
 & & A_{22} & B_{11} & & & C A_{22} B_{21} \\
 A_{11} & B_{22} & & & & & A_{12} B_{22} \\
 & & & & & & A_{22} B_{22} A_{22} B_{21} C A_{12} B_{22} C A_{12} B_{21} \\
 \hline
 A_{11} & B_{11} & & & & & C A_{12} B_{21} ;
 \end{array}$$

que corresponde a la ecuación (4.11).

Del mismo modo, establecemos

$$C_{12} D P_1 C P_2 ;$$

y entonces  $C_{12}$  es igual

$$\begin{array}{ccccc}
 A_{11} & B_{12} & A_{11} & B_{22} & \\
 & C A_{11} & B_{22} & C A_{12} & B_{22} \\
 \hline
 A_{11} & B_{12} & & & C A_{12} B_{22} ;
 \end{array}$$

correspondiente a la ecuación (4.12).

Configuración

$$C_{21} D P_3 C P_4$$

hace que  $C_{21}$  sea igual

$$\begin{array}{ccccc}
 A_{21} & B_{11} & C A_{22} & B_{11} & \\
 & A_{22} & B_{11} & C A_{22} & B_{21} \\
 \hline
 A_{21} & B_{11} & & & C A_{22} B_{21} ;
 \end{array}$$

correspondiente a la ecuación (4.13).

Finalmente, establecemos

$$C_{22} D P_5 C P_1 P_3 P_7 ;$$

para que  $C_{22}$  sea igual

$$\begin{array}{ccccccc}
 A_{11} & B_{11} & C A_{11} & B_{22} & C A_{22} & B_{11} & C A_{22} B_{22} \\
 & A_{11} & B_{22} & & & C A_{11} & B_{12} \\
 & & A_{22} & B_{11} & & & A_{21} B_{11} \\
 A_{11} & B_{11} & & & & A_{11} B_{12} & C A_{21} B_{11} C A_{21} B_{12} \\
 \hline
 & & & & & A_{22} B_{22} & C A_{21} B_{12} ;
 \end{array}$$

que corresponde a la ecuación (4.14). En total, sumamos o restamos matrices  $n=2$   $n=2$  ocho veces en el paso 4, por lo que este paso realmente toma  $,n^2/$  tiempo.

Por lo tanto, vemos que el algoritmo de Strassen, que comprende los pasos 1 a 4, produce el producto de matriz correcto y que la recurrencia (4.18) caracteriza su tiempo de ejecución. Como veremos en la Sección 4.5 que esta recurrencia tiene la solución  $T(n) \leq D \cdot n^{\lg 7}$ , el método de Strassen es asintóticamente más rápido que el procedimiento sencillo MATRIZ CUADRADA -MULTIPLICACIÓN . Las notas al final de este capítulo analizan algunos de los aspectos prácticos del algoritmo de Strassen.

### Ejercicios

Nota: aunque los ejercicios 4.2-3, 4.2-4 y 4.2-5 tratan sobre variantes del algoritmo de Strassen, debe leer la sección 4.5 antes de intentar resolverlos.

#### 4.2-1

Usar el algoritmo de Strassen para calcular el producto de matrices

$$\begin{array}{cc} 1 & 3 \\ 7 & 5 \end{array} \quad \begin{array}{cc} 6 & 8 \\ 4 & 2 \end{array}$$

Muestra tu trabajo.

#### 4.2-2

Escriba el pseudocódigo para el algoritmo de Strassen.

#### 4.2-3

¿Cómo modificaría el algoritmo de Strassen para multiplicar  $nn$  matrices en las que  $n$  no es una potencia exacta de 2? Demuestre que el algoritmo resultante se ejecuta en el tiempo  $,n^{\lg 7}/$ .

#### 4.2-4

¿Cuál es el mayor  $k$  tal que si puede multiplicar 3 3 matrices usando  $k$  multiplicaciones (sin asumir la comutatividad de la multiplicación), entonces puede multiplicar  $nn$  matrices en el tiempo en  $7^k/$ ? ¿Cuál sería el tiempo de ejecución de este algoritmo?

#### 4.2-5

V. Pan ha descubierto una manera de multiplicar 68 68 matrices usando 132 464 multiplicaciones, una manera de multiplicar 70 70 matrices usando 143 640 multiplicaciones y una manera de multiplicar 72 72 matrices usando 155 424 multiplicaciones. ¿Qué método produce el mejor tiempo de ejecución asintótico cuando se usa en un algoritmo de multiplicación de matrices de divide y vencerás? ¿Cómo se compara con el algoritmo de Strassen?

## 4.2-6

¿Qué tan rápido puede multiplicar una matriz  $k \times n$  por una matriz  $n \times n$ , usando el algoritmo de Strassen como subrutina? Responda la misma pregunta con el orden de las matrices de entrada invertido.

## 4.2-7

Muestre cómo multiplicar los números complejos a  $C_b$  y  $C_d$  usando solo tres multiplicaciones de números reales. El algoritmo debe tomar  $a, b, c$  y  $d$  como entrada y producir el componente real  $ac bd$  y el componente imaginario  $ad + bc$  por separado.

### 4.3 El método de sustitución para resolver recurrencias

Ahora que hemos visto cómo las recurrencias caracterizan los tiempos de ejecución de los algoritmos divide y vencerás, aprenderemos a resolver las recurrencias. Comenzamos en esta sección con el método de "sustitución".

El método de sustitución para resolver recurrencias consta de dos pasos:

1. Adivina la forma de la solución.
2. Usa la inducción matemática para encontrar las constantes y mostrar que la solución obras.

Sustituimos la solución adivinada por la función cuando aplicamos la hipótesis inductiva a valores más pequeños; de ahí el nombre de "método de sustitución". Este método es poderoso, pero debemos ser capaces de adivinar la forma de la respuesta para poder aplicarlo.

Podemos usar el método de sustitución para establecer límites superiores o inferiores en una recurrencia. Como ejemplo, determinemos un límite superior en la recurrencia  $T(n) = 2T(n/2) + cn$

$$2T(n) = 2T(n/2) + cn \quad : \quad (4.19)$$

que es similar a las recurrencias (4.3) y (4.4). Suponemos que la solución es  $T(n) \leq cn \lg n$ . El método de sustitución requiere que demosbremos que  $T(n) \leq cn \lg n$  para una elección apropiada de la constante  $c > 0$ . Comenzamos suponiendo que este límite se cumple para todos los  $m < n$  positivos, en particular para  $m = n/2$ , lo que produce  $T(n) \leq cn \lg(n/2) + cn$ . Sustituyendo en la recurrencia se obtiene  $T(n) \leq cn \lg(n/2) + cn \lg(n/2) + cn = cn \lg n$ .

$$2cn \lg(n/2) + cn \leq cn \lg n$$

$$cn \lg(n/2) \leq cn \lg n$$

$$n \lg(n/2) \leq n \lg n$$

:

donde el último paso se mantiene mientras c

1.

La inducción matemática ahora requiere que mostremos que nuestra solución se cumple para las condiciones de contorno. Por lo general, lo hacemos demostrando que las condiciones de contorno son adecuadas como casos base para la prueba inductiva. Para la recurrencia (4.19), debemos demostrar que podemos elegir la constante c lo suficientemente grande como para que la cota  $T \cdot n / cn \lg n$  funcione también para las condiciones de contorno. Este requisito a veces puede dar lugar a problemas. Supongamos, en aras del argumento, que  $T \cdot 1 / D 1$  es la única condición de contorno de la recurrencia. Entonces, para  $n \geq 1$ , el límite  $T \cdot n / cn \lg n$  produce  $T \cdot 1 / c_1 \lg 1 \leq 1$ , lo que está en desacuerdo con  $T \cdot 1 / D 1$ .

En consecuencia, el caso base de nuestra prueba inductiva no se cumple.

Podemos superar este obstáculo probando una hipótesis inductiva para una condición de contorno específica con solo un poco más de esfuerzo. En la recurrencia (4.19), por ejemplo, aprovechamos la notación asintótica que solo nos obliga a probar  $T \cdot n / cn \lg n$  para  $n \geq n_0$ , donde  $n_0$  es una constante que podemos elegir. Mantenemos la problemática condición de contorno  $T \cdot 1 / D 1$ , pero la eliminamos de la consideración en la prueba inductiva. Lo hacemos observando primero que para  $n > 3$ , la recurrencia no depende directamente de  $T \cdot 1 /$ . Así, podemos reemplazar  $T \cdot 1 /$  por  $T \cdot 2 /$  y  $T \cdot 3 /$  como casos base en la prueba inductiva, dejando  $n_0 \geq 4$ . Note que hacemos una distinción entre el caso base de la recurrencia ( $n \geq 1$ ) y los casos base de la prueba inductiva ( $n \geq 2$  y  $n \geq 3$ ). Con  $T \cdot 1 / D 1$ , derivamos de la recurrencia que  $T \cdot 2 / D 4$  y  $T \cdot 3 / D 5$ . Ahora podemos completar la demostración inductiva de que  $T \cdot n / cn \lg n$  para alguna constante  $c > 1$  eligiendo  $c$  lo suficientemente grande como para que  $T \cdot 2 / c_2 \lg 2$  y  $T \cdot 3 / c_3 \lg 3$ . Resulta que cualquier elección de  $c > 1$  es suficiente para que se cumplan los casos base de  $n \geq 2$  y  $n \geq 3$ . Para la mayoría de las recurrencias

Como examinaremos, es sencillo extender las condiciones de contorno para hacer que la suposición inductiva funcione para  $n$  pequeña, y no siempre resolveremos explícitamente los detalles.

#### Haciendo una buena conjectura

Desafortunadamente, no existe una forma general de adivinar las soluciones correctas para las recurrencias. Adivinar una solución requiere experiencia y, ocasionalmente, creatividad. Afortunadamente, sin embargo, puedes usar algunas heurísticas para ayudarte a convertirte en un buen adivinador. También puedes usar árboles de recursión, que veremos en la Sección 4.4, para generar buenas suposiciones.

Si una recurrencia es similar a una que ha visto antes, entonces es razonable adivinar una solución similar. Como ejemplo, considere la recurrencia

$$T \cdot n / D \geq 2T \cdot n + C \cdot 17 / n$$

lo cual parece difícil debido al agregado "17" en el argumento de  $T$  en el lado derecho. Intuitivamente, sin embargo, este término adicional no puede afectar sustancialmente la

solución a la recurrencia. Cuando  $n$  es grande, la diferencia entre  $b_n=2c$  y  $b_n=2c$  C 17 no es tan grande: ambos cortan  $n$  casi por la mitad. En consecuencia, hacemos la conjetura de que  $T_{.n} \approx D \ln n$ , lo cual puede verificarse como correcto usando el método de sustitución (vea el Ejercicio 4.3-6).

Otra forma de hacer una buena conjetura es demostrar límites superiores e inferiores flexibles en la recurrencia y luego reducir el rango de incertidumbre. Por ejemplo, podríamos comenzar con una cota inferior de  $T_{.n} \geq D_{.n}$  para la recurrencia (4.19), ya que tenemos el término  $n$  en la recurrencia, y podemos probar una cota superior inicial de  $T_{.n} \leq D \ln 2^n$ . Luego, podemos disminuir gradualmente el límite superior y aumentar el límite inferior hasta que converjamos en la solución asintóticamente ajustada correcta de  $T_{.n} \approx D_{.n} \ln n$ .

### sutilezas

A veces, puede adivinar correctamente un límite asintótico en la solución de una recurrencia, pero de alguna manera las matemáticas no funcionan en la inducción. El problema frecuentemente resulta ser que la suposición inductiva no es lo suficientemente fuerte para probar el límite detallado. Si revisa la conjetura restando un término de orden inferior cuando se encuentra con un obstáculo de este tipo, las matemáticas a menudo pasan.

Considere la recurrencia

$$T_{.n} = DT_{.n-1} + b_n = CT_{.n-1} + d_n$$

Suponemos que la solución es  $T_{.n} \approx D \ln n$ , y tratamos de demostrar que  $T_{.n} \approx cn$  para una elección adecuada de la constante  $c$ . Sustituyendo nuestra conjetura en la recurrencia, obtenemos

$$\begin{aligned} T_{.n} &= D \ln n + b_n \\ &\approx D \ln n + cn \end{aligned}$$

lo que no implica  $T_{.n} \approx cn$  para cualquier elección de  $c$ . Podríamos tener la tentación de intentar una conjetura mayor, digamos  $T_{.n} \approx D \ln n + 2$ . Aunque podemos hacer que esta conjetura mayor funcione, nuestra conjetura original de  $T_{.n} \approx D \ln n$  es correcta. Sin embargo, para demostrar que es correcta, debemos hacer una hipótesis inductiva más fuerte.

Intuitivamente, nuestra conjetura es casi correcta: estamos equivocados solo por la constante 1, un término de orden inferior. Sin embargo, la inducción matemática no funciona a menos que demostremos la forma exacta de la hipótesis inductiva. Superamos nuestra dificultad restando un término de orden inferior de nuestra conjetura anterior. Nuestra nueva conjetura es  $T_{.n} \approx cn + d$ , donde  $d \approx 0$  es una constante. ahora tenemos

$$\begin{aligned} T_{.n} &= D \ln n + b_n + cn + d \\ &\approx D \ln n + cn + cn \end{aligned}$$

siempre que  $d > 1$ . Como antes, debemos elegir la constante  $c$  lo suficientemente grande como para manejar las condiciones de contorno.

Puede encontrar la idea de restar un término de orden inferior contradictorio. Después de todo, si las matemáticas no funcionan, debemos aumentar nuestra suposición, ¿verdad? ¡No necesariamente! Al probar un límite superior por inducción, en realidad puede ser más difícil demostrar que se cumple un límite superior más débil, porque para probar el límite más débil, debemos usar el mismo límite más débil inductivamente en la prueba.

En nuestro ejemplo actual, cuando la recurrencia tiene más de un término recursivo, podemos restar el término de orden inferior del límite propuesto una vez por término recursivo. En el ejemplo anterior, restamos la constante  $d$  dos veces, una para el término  $T_{n+1} \leq 2c$  y otra para el término  $T_n \leq 2e$ . Terminamos con la desigualdad  $T_n \leq cn^2$ , y fue fácil encontrar valores de  $d$  para hacer que  $cn^2$  sea menor o igual que  $cn^d$ .

### Evitar trampas

Es fácil equivocarse en el uso de la notación asintótica. Por ejemplo, en la recurrencia (4.19) podemos “probar” falsamente  $T_n \leq D$  dividiendo  $T_n$  entre  $cn$  y luego argumentando

$$\begin{aligned} T_n &\leq 2c n^2 = 2c / C n \\ &\leq cn^2 \text{ norte} \\ D \text{ Encendido} &; \quad \text{¡equivocado!!} \end{aligned}$$

ya que  $c$  es una constante. El error es que no hemos probado la forma exacta de la hipótesis inductiva, es decir, que  $T_n \leq cn^2$ . Por lo tanto, probaremos explícitamente que  $T_n \leq cn^2$  cuando queramos demostrar que  $T_n \leq D$ .

### Cambio de variables

A veces, un poco de manipulación algebraica puede hacer que una recurrencia desconocida sea similar a una que haya visto antes. Como ejemplo, considere la recurrencia

$$T_n \leq D + 2T_m + C \lg n$$

que parece difícil. Sin embargo, podemos simplificar esta recurrencia con un cambio de variables. Por comodidad, no nos preocuparemos por redondear valores, como  $m$ , para que sean números enteros. Cambiar el nombre de  $m$  a  $\lg n$

$$\text{produce } T_m \leq D + 2T_{m-1} + C m$$

Ahora podemos renombrar  $S_m = DT_m + Cm$  para producir la nueva recurrencia

$$S_m \leq D + 2S_{m-1}$$

que es muy parecido a la recurrencia (4.19). En efecto, esta nueva recurrencia tiene la misma solución:  $S_m / D \Omega m \lg m /$ . Cambiando de nuevo de  $S_m / a T .n /$ , obtenemos

$T .n / DT .2m / D Sm / D \Omega m \lg m / D O.\lg n \lg \lg n / :$

### Ejercicios

#### 4.3-1

Demuestre que la solución de  $T .n / DT .n 1 / C n$  es  $O.n^2 /$ .

#### 4.3-2

Muestre que la solución de  $T .n / DT .dn=2e / C 1$  es  $O.\lg n /$ .

#### 4.3-3

Vimos que la solución de  $T .n / D 2T .bn=2c/Cn$  es  $O.n \lg n /$ . Demuestre que la solución de esta recurrencia también es  $.n \lg n /$ . Concluya que la solución es  $.n \lg n /$ .

#### 4.3-4

Muestre que al hacer una hipótesis inductiva diferente, podemos superar la dificultad con la condición límite  $T .1 / D 1$  para la recurrencia (4.19) sin ajustar las condiciones límite para la prueba inductiva.

#### 4.3-5

Muestre que  $.n \lg n /$  es la solución a la recurrencia "exacta" (4.3) para la ordenación por fusión.

#### 4.3-6

Demuestre que la solución de  $T .n / D 2T .bn=2c C 17 / C n$  es  $O.n \lg n /$ .

#### 4.3-7

Utilizando el método maestro de la Sección 4.5, puede demostrar que la solución a la recurrencia  $T .n / D 4T .n=3 / C n$  es  $T .n / D ,n\log_3 4 /$ . Demuestre que una prueba de sustitución  $\log_3 4$  supuesto  $T .n /$  falla. Luego muestra cómo restar un término de orden inferior para hacer que funcione una prueba de sustitución.

#### 4.3-8

Utilizando el método maestro de la Sección 4.5, puede demostrar que la solución a la recurrencia  $T .n / D 4T .n=2 / C n^2$  es  $T .n / D ,n^2 /$ . Demuestre que una sustitución  $\log_2 4$  falla. Luego muestre de orden inferior para hacer que una prueba con el supuesto  $T .n /$  funcione. Luego muestra cómo restar una prueba con el supuesto  $T .n /$  para hacer que una prueba de sustitución funcione.

## 4.3-9

Resuelva la recurrencia  $T(n) = 3T(n/4) + C \log n$  haciendo un cambio de variables.

Su solución debe ser asintóticamente ajustada. No te preocupes por si los valores son integrales.

#### 4.4 El método del árbol de recurrencias para resolver recurrencias

Aunque puede usar el método de sustitución para proporcionar una prueba sucinta de que la solución a una recurrencia es correcta, es posible que tenga problemas para encontrar una buena suposición. Dibujar un árbol de recurrencia, como hicimos en nuestro análisis de la recurrencia del ordenamiento por fusión en la Sección 2.3.2, sirve como una forma directa de idear una buena suposición. En un árbol recursivo, cada nodo representa el costo de un solo subproblema en algún lugar del conjunto de invocaciones de funciones recursivas. Sumamos los costos dentro de cada nivel del árbol para obtener un conjunto de costos por nivel y luego sumamos todos los costos por nivel para determinar el costo total de todos los niveles de la recursividad.

Un árbol de recurrencia se usa mejor para generar una buena suposición, que luego puede verificar mediante el método de sustitución. Al usar un árbol de recurrencia para generar una buena suposición, a menudo puede tolerar una pequeña cantidad de "descuido", ya que verificará su suposición más adelante. Sin embargo, si tiene mucho cuidado al dibujar un árbol de recurrencia y sumar los costos, puede usar un árbol de recurrencia como prueba directa de una solución a una recurrencia. En esta sección, usaremos árboles de recurrencia para generar buenas conjeturas, y en la Sección 4.6, usaremos árboles de recurrencia directamente para probar el teorema que forma la base del método maestro.

Por ejemplo, veamos cómo un árbol de recurrencia proporcionaría una buena suposición para la recurrencia  $T(n) = 3T(n/4) + cn^2$ . Comenzamos enfocándonos en encontrar un límite superior para la solución. Como sabemos que los pisos y los techos generalmente no importan cuando se resuelven recurrencias (aquí hay un ejemplo de descuido que podemos tolerar), creamos un árbol de recurrencia para la recurrencia  $T(n) = 3T(n/4) + cn^2$ , habiendo escrito el coeficiente constante implícito  $c > 0$ .

La figura 4.5 muestra cómo derivamos el árbol de recurrencia para  $T(n) = 3T(n/4) + cn^2$ . Por conveniencia, asumimos que  $n$  es una potencia exacta de 4 (otro ejemplo de descuido tolerable) de manera que todos los tamaños de los subproblemas son números enteros. La parte (a) de la figura muestra  $T(n)$ , que expandimos en la parte (b) en un árbol equivalente que representa la recurrencia. El término  $cn^2$  en la raíz representa el costo en el nivel superior de recursión, y los tres subárboles de la raíz representan los costos incurridos por los subproblemas de tamaño  $n=4$ . La parte (c) muestra que este proceso se llevó un paso más allá al expandir cada nodo con un costo  $T(n/4)$  de la parte (b). El costo de cada uno de los tres hijos de la raíz es  $cn^2/4$ . Continuamos expandiendo cada nodo en el árbol dividiéndolo en sus partes constituyentes según lo determine la recurrencia.

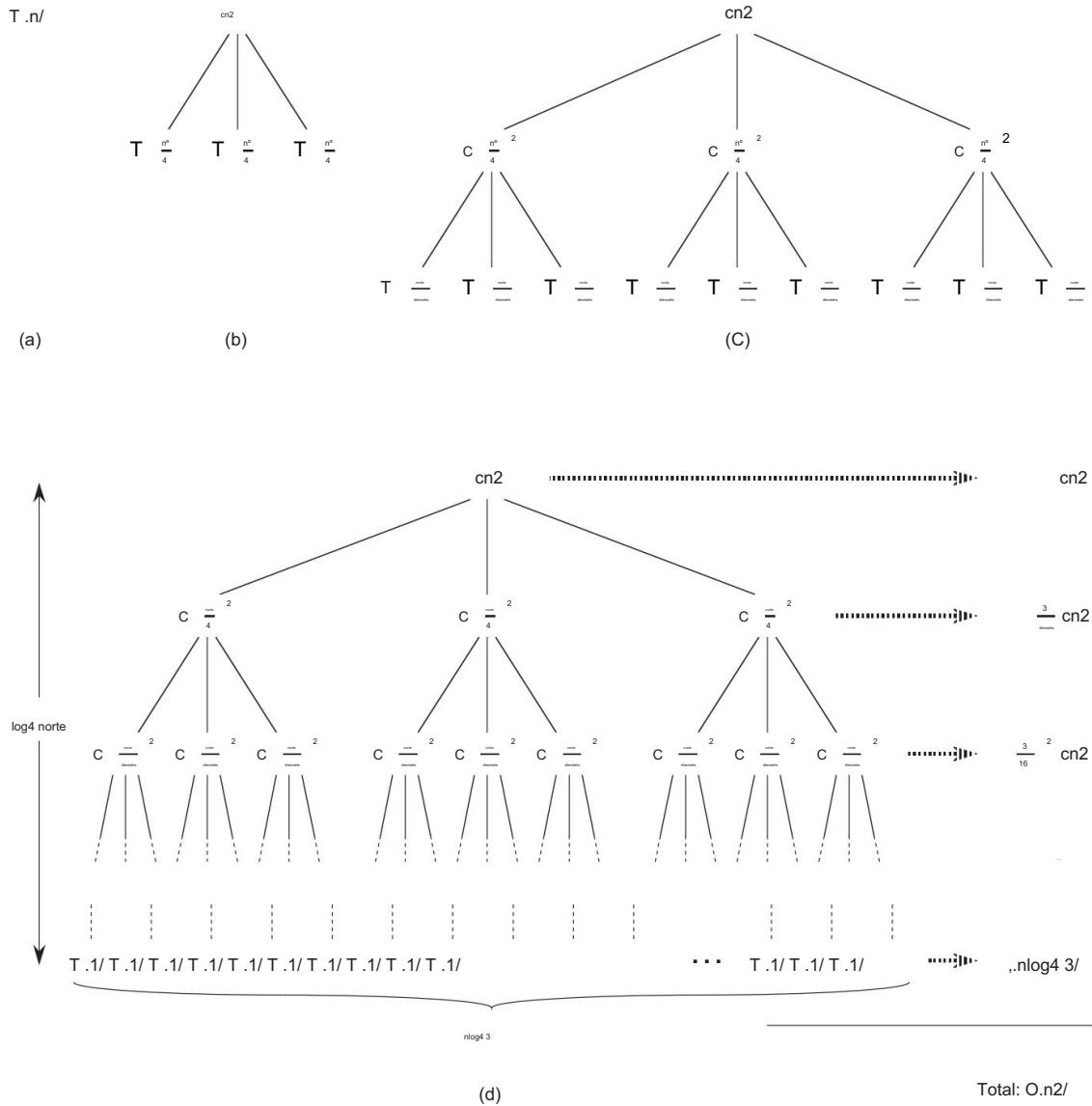


Figura 4.5 Construcción de un árbol de recurrencia para la recurrencia  $T.n/ \leq 3T .n=4/ + cn2$ . La parte (a) muestra  $T.n/$ , que se expande progresivamente en (b)–(d) para formar el árbol de recurrencia. El árbol completamente expandido en la parte (d) tiene una altura  $\log_4 n$  (tiene  $\log_4 n$  C 1 niveles).

Debido a que los tamaños de los subproblemas disminuyen por un factor de 4 cada vez que bajamos un nivel, eventualmente debemos alcanzar una condición límite. ¿A qué distancia de la raíz llegamos a uno? El tamaño del subproblema para un nodo en la profundidad  $i$  es  $n=4^i$ . Por lo tanto, el tamaño del subproblema llega a  $n \leq D$  cuando  $n=4^i \leq D$ , o, de manera equivalente, cuando  $i \leq \log_4 D$ . Así, el árbol tiene  $\log_4 n$  niveles (a profundidades 0; 1; 2; ; ; ;  $\log_4 n$ ).

A continuación determinamos el costo en cada nivel del árbol. Cada nivel tiene tres veces más nodos que el nivel superior, por lo que el número de nodos en la profundidad  $i$  es  $3^i$ . Debido a que los tamaños de los subproblemas se reducen por un factor de 4 para cada nivel, descendemos desde la raíz, cada nodo en la profundidad  $i$ , para  $i \in \{0; 1; 2; \dots; \log_4 n\}$ , tiene un costo de  $c_n = 4^i / 2$ . Multiplicando, vemos que el costo total sobre todos los nodos en la profundidad  $i$ , para  $i \in \{0; 1; 2; \dots; \log_4 n\}$ , es  $3^i c_n = 4^i / 2 \cdot 3^i = 16^i / 2^n$ . El nivel inferior, a la profundidad  $\log_4 n$ , tiene  $3 \log_4 n$  nodos, cada uno de los cuales contribuye con un costo  $T_{1/1}$ , para un costo total de  $n \log_4 3 T_{1/1}$ , que es  $n \log_4 3 / 2^n$ , ya que suponemos que  $T_{1/1}$  es una constante.

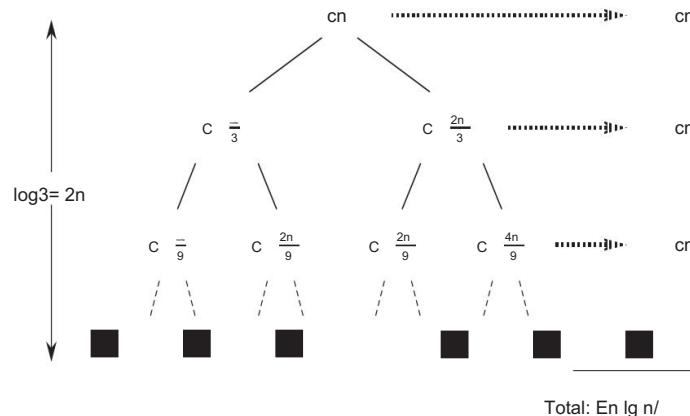
Ahora sumamos los costos de todos los niveles para determinar el costo de todo el árbol:

$$\begin{array}{l}
 T \cdot n / D \text{ cn2 C} \quad \frac{3}{16} \text{ cn2 C} \quad \overline{3} \cdot 16^2 \text{ cn2 CC} \cdot 3 \cdot 16^1 \quad \overline{3} \cdot 16^{\log_4 n} \cdot n^1 \text{ cn2 C} \cdot \dots \cdot n^{\log_4 3} \\
 \text{registro } n^1 \\
 \text{D} \quad X4 \quad \overline{\text{cn2 C} \cdot \dots \cdot n^{\log_4 3}} \\
 \text{iD0} \\
 \text{D} \quad \frac{.3 \cdot 16^{\log_4 1}}{n \cdot .3 \cdot 16^1} \text{ cn2 C} \cdot \dots \cdot n^{\log_4 3} / \quad (\text{por la ecuación (A.5)) :})
 \end{array}$$

Esta última fórmula parece algo desordenada hasta que nos damos cuenta de que nuevamente podemos aprovechar pequeñas cantidades de descuido y usar una serie geométrica decreciente infinita como límite superior. Retrocediendo un paso y aplicando la ecuación (A.6), tenemos

$$\begin{aligned}
 & T . n/D \\
 & \text{registro } n1 \\
 & X4 \quad \overline{3} \ 16i \ cn2 \ C , .nlog4 \ 3/ \\
 & iD0 \\
 \\
 & < X1 \quad \overline{3} \ 16i \ cn2 \ C , .nlog4 \ 3/ \\
 & iD0 \\
 & D \quad \overline{1} \quad cn2 \ C , .nlog4 \ 3/ \ 1 \\
 & .3=16/ \ 16 \\
 \\
 & D \quad \overline{cn2 \ C , .nlog4 \ 3 / 13} \ D
 \end{aligned}$$

Por lo tanto, hemos derivado una conjetura de  $T(n)/D(n^2)$  para nuestra recurrencia original  $T(n) = 3T(\frac{n}{4}) + cn^2$ . En este ejemplo, los coeficientes de  $c n^2$  forman una serie geométrica decreciente y, por la ecuación (A.6), la suma de estos coeficientes

Figura 4.6 Un árbol de recurrencia para la recurrencia  $T(n) = 3T(n/3) + cn$ .

está acotado superiormente por la constante  $16=13$ . Dado que la contribución de la raíz al costo total es  $cn^2$ , la raíz contribuye con una fracción constante del costo total. En otras palabras, el costo de la raíz domina el costo total del árbol.

De hecho, si  $O(n^2)$  es de hecho un límite superior para la recurrencia (como verificaremos en un momento), entonces debe ser un límite estrecho. ¿Por qué? La primera llamada recursiva contribuye con un costo de  $n^2$ , por lo que  $n^2$  debe ser un límite inferior para la recurrencia.

Ahora podemos usar el método de sustitución para verificar que nuestra estimación fue correcta, es decir,  $T(n) = O(n^2)$  es un límite superior para la recurrencia  $T(n) = 3T(n/3) + cn^2$ . Queremos mostrar que  $T(n) \leq d n^2$  para alguna constante  $d > 0$ . Usando la misma constante  $c > 0$  que antes, tenemos

$$\begin{aligned}
 T(n) &= 3T(n/3) + cn^2 \\
 &\leq 3d(n/3)^2 + cn^2 \\
 &= 3d \cdot \frac{n^2}{9} + cn^2 \\
 &= \frac{3}{16}dn^2 + cn^2 \\
 &\leq dn^2 ;
 \end{aligned}$$

donde el último paso se mantiene mientras  $d \cdot 16 \geq c$ .

En otro ejemplo más complejo, la figura 4.6 muestra el árbol de recurrencia para

$T(n) = 3T(n/3) + cn^2$  Encendido/ :

(De nuevo, omitimos las funciones suelo y techo por simplicidad.) Como antes, hacemos que  $c$  represente el factor constante en el término  $On^2$ . Cuando sumamos los valores de los niveles del árbol de recurrencia que se muestra en la figura, obtenemos un valor de  $cn$  para cada nivel.

¡El camino simple más largo desde la raíz hasta una hoja es  $n! \cdot 2=3/n! \cdot 2=3/2n$  !

! 1. Como  $.2=3/kn$  D 1 cuando  $k D \log_3=2 n$ , la altura del árbol es  $\log_3=2 n$ . Intuitivamente, esperamos que la solución a la recurrencia sea como máximo el número de niveles multiplicado por el costo de cada nivel, o  $O.cn \log_3=2 n / D On \lg n$ . Sin embargo, la figura 4.6 muestra solo los niveles superiores del árbol de recurrencia y no todos los niveles del árbol contribuyen con un costo de  $cn$ . Considere el costo de las hojas. Si este árbol fuera un árbol binario completo de altura  $\log_3=2 n$ , habría  $2\log_3=2 n$  hojas. Dado recursivo D  $n\log_3=2 2$  que el costo de cada hoja es una constante, el costo total de todas las hojas sería entonces  $.n\log_3=2 2$  que, dado que  $\log_3=2 2$  es una constante estrictamente mayor que 1, es  $!n \lg n$ . Sin embargo, este árbol de recurrencia no es un árbol binario completo, por lo que tiene menos de  $n\log_3=2 2$  hojas. Además, a medida que descendemos desde la raíz, cada vez faltan más nodos internos. En consecuencia, los niveles hacia la parte inferior del árbol recursivo contribuyen menos que  $cn$  al costo total. Podríamos elaborar una contabilidad precisa de todos los costos, pero recuerde que solo estamos tratando de obtener una conjectura para usar en el método de sustitución. Toleremos el descuido e intentemos demostrar que una conjectura de  $On \lg n$  para el límite superior es correcta.

De hecho, podemos usar el método de sustitución para verificar que  $On \lg n$  es un límite superior para la solución de la recurrencia. Mostramos que  $T .n/ dn \lg n$ , donde  $d$  es una constante positiva adecuada. Tenemos

$$\begin{aligned} T .n/ &= T .n=3/ CT .2n=3/ C cn dn=3/ \\ &\quad lg.n=3/ C d.2n=3/lg.2n=3/ C cn D .dn=3/lg n \\ &\quad dn=3 /lg 3/ C .d.2n=3/lg n d.2n=3/ \\ &\quad \quad lg.3=2// C cn D dn lg n d..n=3/lg 3 C .2n=3/lg. \\ &\quad 3=2// C cn D dn lg n d..n=3/lg 3 C .2n=3/lg 3 .2n=3/lg 2/ C \\ &\quad cn D dn lg nd n.lg 3 2=3/ Cn dn lg n \end{aligned}$$

siempre que  $dc=.lg 3 .2=3//$ . Por lo tanto, no necesitábamos realizar una contabilidad más precisa de los costos en el árbol de recurrencia.

### Ejercicios

#### 4.4-1

Use un árbol de recurrencia para determinar un buen límite superior asintótico en la recurrencia  $T .n/ D 3T .bn=2c/ C n$ . Usa el método de sustitución para verificar tu respuesta.

#### 4.4-2

Use un árbol de recurrencia para determinar un buen límite superior asintótico en la recurrencia  $T .n/ DT .n=2/ C n^2$ . Usa el método de sustitución para verificar tu respuesta.

## 4.4-3

Use un árbol de recurrencia para determinar un buen límite superior asintótico en la recurrencia  $T .n/ D 4T .n=2 C 2/ C n$ . Usa el método de sustitución para verificar tu respuesta.

## 4.4-4

Usa un árbol de recurrencia para determinar un buen límite superior asintótico en la recurrencia  $T .n/ D 2T .n 1/ C 1$ . Usa el método de sustitución para verificar tu respuesta.

## 4.4-5

Utilice un árbol de recurrencia para determinar un buen límite superior asintótico en la recurrencia  $T .n/ DT .n1/CT .n=2/Cn$ . Usa el método de sustitución para verificar tu respuesta.

## 4.4-6

Argumente que la solución a la recurrencia  $T .n/ DT .n=3/CT .2n=3/Ccn$ , donde c es una constante, es  $n \lg n$  apelando a un árbol de recurrencia.

## 4.4-7

Dibuje el árbol de recurrencia para  $T .n/ D 4T .bn=2c/ C cn$ , donde c es una constante, y proporcione un límite asintótico estrecho en su solución. Verifique su límite por el método de sustitución.

## 4.4-8

Use un árbol de recurrencia para dar una solución asintóticamente ajustada a la recurrencia  $T .n/ DT .na/ CT .a/ C cn$ , donde a 1 y  $c>0$  son constantes.

## 4.4-9

Use un árbol de recurrencia para dar una solución asintóticamente ajustada a la recurrencia  $T .n/ DT .n/ CT ..1 /n/ C cn$ , donde  $c$  es una constante en el rango  $0 < c < 1$  y  $c > 0$  también es una constante.

## 4.5 El método maestro para resolver recurrencias

El método maestro proporciona un método de "libro de recetas" para resolver recurrencias de la forma

$$T .n/ D aT .n=b/ C f .n/ ; \quad (4.20)$$

donde  $a \geq 1$  y  $b > 1$  son constantes y  $f .n/$  es una función asintóticamente positiva. Para usar el método maestro, deberá memorizar tres casos, pero luego podrá resolver muchas recurrencias con bastante facilidad, a menudo sin lápiz ni papel.

La recurrencia (4.20) describe el tiempo de ejecución de un algoritmo que divide un problema de tamaño  $n$  en subproblemas, cada uno de tamaño  $n=b$ , donde  $a$  y  $b$  son constantes positivas. Los subproblemas a se resuelven recursivamente, cada uno en el tiempo  $T(n=b)$ . La función  $f(n)$  abarca el costo de dividir el problema y combinar los resultados de los subproblemas. Por ejemplo, la recurrencia que surge del algoritmo de Strassen tiene a  $D(7)$ ,  $b D(2)$  y  $f(n)=D(n^2)$ .

Como cuestión de corrección técnica, la recurrencia no está realmente bien definida, porque  $n=b$  podría no ser un número entero. Sin embargo, reemplazar cada uno de los términos a  $T(n=b)$  con  $T(bn=bc)$  o  $T(dn=be)$  no afectará el comportamiento asintótico de la recurrencia. (Probaremos esta afirmación en la siguiente sección.) Normalmente encontramos conveniente, por lo tanto, omitir las funciones de suelo y techo cuando escribimos recurrencias de divide y vencerás de esta forma.

### El teorema del maestro

El método maestro depende del siguiente teorema.

#### Teorema 4.1 (Teorema del maestro)

Sean  $a \geq 1$  y  $b > 1$  constantes, sea  $f(n)$  una función, y defina  $T(n)$  en los enteros no negativos por la recurrencia

$$T(n) = aT(n/b) + f(n);$$

donde interpretamos que  $n=b$  significa  $bn=bc$  o  $dn=be$ . Entonces  $T(n)$  tiene los siguientes límites asintóticos:

1. Si  $f(n) = O(n \log b/a)$  para alguna constante  $> 0$ , entonces  $T(n) = O(n \log b/a)$ .
2. Si  $f(n) = D(n \log b/a)$ , entonces  $T(n) = D(n \log b/a \lg n)$ .
3. Si  $f(n) = D(n \log b/a) + c$  para alguna constante  $> 0$ , y si  $a \cdot n = b^k$  para alguna constante  $c < 1$  y todas  $n$  suficientemente grandes, entonces  $T(n) = f(n)$ . ■

Antes de aplicar el teorema maestro a algunos ejemplos, dediquemos un momento a tratar de entender lo que dice. En cada uno de los tres casos, comparamos la función  $f(n)$  con la función  $n \log b/a$ . Intuitivamente, la mayor de las dos funciones determina la solución a la recurrencia. Si, como en el caso 1, la función  $n \log b/a$  es mayor, entonces la solución es  $T(n) = O(n \log b/a)$ . Si, como en el caso 3, la función  $f(n)$  es mayor, entonces la solución es  $T(n) = f(n)$ . Si, como en el caso 2, las dos funciones son del mismo tamaño, multiplicamos por un factor logarítmico y la solución es  $T(n) = D(n \log b/a \lg n)$ .

Más allá de esta intuición, debe ser consciente de algunos tecnicismos. En el primer caso, no solo  $f(n)$  debe ser menor que  $n \log b/a$ , sino que debe ser polinomialmente menor.

Es decir,  $f(n)$  debe ser asintóticamente menor que  $n \log b$  a por un factor de  $n$  para alguna constante  $>0$ . En el tercer caso, no solo  $f(n)$  debe ser mayor que  $n \log b$ , sino que también debe ser polinomialmente mayor y además satisfacer la condición de "regularidad" de que  $a f(n) = b f(n/2)$ . Esta condición es satisfecha por la mayoría de las funciones acotadas polinómicamente que encontraremos.

Tenga en cuenta que los tres casos no cubren todas las posibilidades para  $f(n)$ . Hay una brecha entre los casos 1 y 2 cuando  $f(n)$  es menor que  $n \log b$  pero no polinomialmente menor. De manera similar, existe una brecha entre los casos 2 y 3 cuando  $f(n)$  es mayor que  $n \log b$  pero no polinomialmente mayor. Si la función  $f(n)$  cae en uno de estos espacios, o si la condición de regularidad en el caso 3 no se cumple, no puede usar el método maestro para resolver la recurrencia.

### Usando el método maestro

Para usar el método maestro, simplemente determinamos qué caso (si corresponde) del teorema maestro se aplica y escribimos la respuesta.

Como primer ejemplo, considere

$$T(n) = 9T(n/3) + Cn$$

Para esta recurrencia, tenemos  $a = 9$ ,  $b = 3$ ,  $f(n) = n$ , y por lo tanto tenemos que  $n \log b = 3 \log 3$  ( $\approx 9.5$ ). Como  $f(n) = O(n \log 3)$ , donde  $D = 1$ , podemos aplicar el caso 1 del teorema maestro y concluir que la solución es  $T(n) = n^2$ .

Ahora considera

$$T(n) = DT(n/2) + Cn$$

en la que  $a = 1$ ,  $b = 2$ ,  $f(n) = n$  y  $n \log b = 1 \log 2 = 1$ . Se aplica el caso 2, ya que  $f(n) = O(n \log b)$ , donde  $D = 1$ , por lo que la solución a la recurrencia es  $T(n) = n \lg n$ .

Por la recurrencia

$$T(n) = 3T(n/3) + Cn \lg n$$

tenemos  $a = 3$ ,  $b = 3$ ,  $f(n) = n \lg n$ , y  $n \log b = 3 \log 3 = 3$ . Como  $f(n) = O(n \log 3)$ , donde  $D = 3$ , se aplica el caso 3 si podemos demostrar que la condición de regularidad se cumple para  $f(n)$ . Para  $n$  suficientemente grande, tenemos que  $a f(n) = b f(n/3)$  ( $\approx 3^2 n \lg n / 3 = 3n \lg n$ ) para  $c = 3$ . En consecuencia, por el caso 3, la solución a la recurrencia es  $T(n) = n \lg n$ .

El método maestro no se aplica a la recurrencia.

$$T(n) = 2T(n/2) + Cn \lg n$$

aunque parece tener la forma adecuada:  $a = 2$ ,  $b = 2$ ,  $f(n) = n \lg n$ , y  $n \log b = 2$ . Podría pensar erróneamente que el caso 3 debería aplicarse, ya que

$f(n) / D \approx n \lg n$  es asintóticamente mayor que  $n \log b \approx D n$ . El problema es que no es polinomialmente mayor. La razón  $f(n) = n \log b \approx D n \lg n = n D \lg n$  es asintóticamente menor que  $n$  para cualquier constante positiva. En consecuencia, la recurrencia cae en la brecha entre el caso 2 y el caso 3. (Vea el Ejercicio 4.6-2 para una solución).

Usemos el método maestro para resolver las recurrencias que vimos en las Secciones 4.1 y 4.2. Recurrencia (4.7),

$$T(n) / D \approx 2T(n/2) + C \cdot n / ;$$

caracteriza los tiempos de ejecución del algoritmo divide y vencerás tanto para el problema del subarreglo máximo como para la ordenación por fusión. (Como es nuestra práctica, omitimos declarar el caso base en la recurrencia.) Aquí, tenemos  $a = 2$ ,  $b = 2$ ,  $f(n) / D \approx n$ , y por lo tanto tenemos que  $\log b \approx \log 2 \approx 1$ . Se aplica el caso 2, ya que  $f(n) / D \approx n$ , por lo que tenemos la solución  $T(n) / D \approx n \lg n$ .

Recurrencia (4.17),

$$T(n) / D \approx 8T(n/2) + C \cdot n^2 / ;$$

describe el tiempo de ejecución del primer algoritmo divide y vencerás que vimos para la multiplicación de matrices. Ahora tenemos  $a = 8$ ,  $b = 2$  y  $f(n) / D \approx n^2$ , por lo que  $\log b \approx \log 2 \approx 1$ . Dado que  $n^3$  es polinomialmente mayor que  $f(n)$  (es decir,  $f(n) / D \approx O(n^3)$  para  $D = 1$ ), se aplica el caso 1 y  $T(n) / D \approx n^3$ .

Finalmente, considere la recurrencia (4.18),

$$T(n) / D \approx 7T(n/2) + C \cdot n^2 / ;$$

que describe el tiempo de ejecución del algoritmo de Strassen. Aquí tenemos  $a = 7$ ,  $b = 2$ ,  $f(n) / D \approx n^2$ , y por lo tanto  $\log b \approx \log 2 \approx 1$ . Reescribiendo  $\log 2 \approx 1$  como  $\lg 2 \approx 1$  y recordando que  $2^{1.8} < \lg 2 < 2^{1.9}$ , vemos que  $f(n) / D \approx O(n \lg 2) \approx O(n)$  para  $D = 1$ .

Nuevamente, se aplica el caso 1 y tenemos la solución  $T(n) / D \approx n \lg 2$ .

## Ejercicios

### 4.5-1

Use el método maestro para dar límites asintóticos ajustados para la siguiente recurrencia

- a.  $T(n) / D \approx 2T(n/4) + C \cdot 1$ .
- b.  $T(n) / D \approx 2T(n/4) + C \cdot n$ .
- c.  $T(n) / D \approx 2T(n/4) + C \cdot n$ .
- d.  $T(n) / D \approx 2T(n/4) + C \cdot n^2$ .

## 4.5-2

El profesor Caesar desea desarrollar un algoritmo de multiplicación de matrices que sea asintóticamente más rápido que el algoritmo de Strassen. Su algoritmo usará el método divide y vencerás, dividiendo cada matriz en partes de tamaño  $n=4$ , y los pasos de dividir y combinar juntos tomarán  $\sqrt{n}$  tiempo. Necesita determinar cuántos subproblemas tiene que crear su algoritmo para vencer al algoritmo de Strassen. Si su algoritmo crea subproblemas, entonces la recurrencia para el tiempo de ejecución  $T(n)$  se convierte en  $T(n) = 4T(\frac{n}{4}) + C\sqrt{n}$ . ¿Cuál es el valor entero más grande de  $a$  para el cual el algoritmo del profesor Caesar sería asintóticamente más rápido que el algoritmo de Strassen?

## 4.5-3

Use el método maestro para mostrar que la solución a la recurrencia de búsqueda binaria  $T(n) = T(\frac{n}{2}) + C\lg n$  es  $T(n) = C\lg n$ . (Consulte el Ejercicio 2.3-5 para obtener una descripción de la búsqueda binaria).

## 4.5-4

¿Se puede aplicar el método maestro a la recurrencia  $T(n) = 4T(\frac{n}{2}) + Cn^2 \lg n$ ?

¿Por qué o por qué no? Proporcione un límite superior asintótico para esta recurrencia.

## 4.5-5 ?

Considere la condición de regularidad  $af(n) \leq cn^k$  para alguna constante  $c < 1$ , que es parte del caso 3 del teorema maestro. Dé un ejemplo de constantes  $a$  y  $b > 1$  y una función  $f(n)$  que satisfaga todas las condiciones en el caso 3 del teorema maestro excepto la condición de regularidad.

## 4.6 Prueba del teorema del maestro

Esta sección contiene una demostración del teorema maestro (Teorema 4.1). No necesitas entender la prueba para aplicar el teorema maestro.

La prueba aparece en dos partes. La primera parte analiza la recurrencia maestra (4.20), bajo el supuesto simplificador de que  $T(n)$  se define sólo sobre potencias exactas de  $b > 1$ , es decir, para  $n = b^k$ ;  $b^k+1$ ;  $b^k+2$ ; ... . Esta parte da toda la intuición necesaria para entender por qué el teorema maestro es verdadero. La segunda parte muestra cómo extender el análisis a todos los enteros positivos  $n$ ; aplica técnica matemática al problema del manejo de pisos y techos.

En esta sección, algunas veces abusaremos ligeramente de nuestra notación asintótica al usarla para describir el comportamiento de funciones que se definen solo sobre potencias exactas de  $b$ .

Recuerde que las definiciones de notaciones asintóticas requieren que

se demuestren límites para todos los números suficientemente grandes, no sólo para aquellos que son potencias de b. Dado que podríamos hacer nuevas notaciones asintóticas que se aplican solo al conjunto  $\{1, 2, 4, 8, \dots\}$ , en lugar de a los números no negativos, este abuso es menor.

Sin embargo, siempre debemos estar en guardia cuando usamos la notación asintótica en un dominio limitado para no sacar conclusiones incorrectas. Por ejemplo, probar que  $T(n) \leq Cn^2$  cuando n es una potencia exacta de 2 no garantiza que  $T(n) \leq Cn^2$ .

La función  $T(n)$  podría definirse como

$$T(n) = \begin{cases} n^2 & \text{si } n \in \{1, 2, 4, 8, \dots\}; \\ \text{lo contrario} & \text{en otro caso.} \end{cases}$$

en cuyo caso el mejor límite superior que se aplica a todos los valores de n es  $T(n) \leq Cn^2$ . Debido a este tipo de consecuencia drástica, nunca utilizaremos la notación asintótica sobre un dominio limitado sin dejar absolutamente claro por el contexto que lo estamos haciendo.

#### 4.6.1 La prueba de potencias exactas

La primera parte de la demostración del teorema maestro analiza la recurrencia (4.20)

$$T(n) \leq aT(n/b) + C \quad \text{para } n \geq 1;$$

para el método maestro, bajo el supuesto de que n es una potencia exacta de  $b > 1$ , donde b no necesita ser un número entero. Desglosamos el análisis en tres lemas. El primero reduce el problema de resolver la recurrencia maestra al problema de evaluar una expresión que contiene una sumatoria. El segundo determina los límites de esta suma. El tercer lema une los dos primeros para probar una versión del teorema maestro para el caso en que n es una potencia exacta de b.

#### Lema 4.2

Sean  $a \geq 1$  y  $b > 1$  constantes, y sea  $f(n)$  una función no negativa definida sobre potencias exactas de b. Defina  $T(n)$  en potencias exactas de b por la recurrencia

$$\text{si } n \geq 1$$

$$T(n) = \begin{cases} aT(n/b) + f(n/b) & \text{si } n \geq 1 \\ 0 & \text{en otro caso.} \end{cases}$$

donde i es un entero positivo. Entonces

$$T(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(b^j) \quad (4.21)$$

Prueba Usamos el árbol de recurrencia en la Figura 4.7. La raíz del árbol ha costado  $f(n)$ , y tiene hijos, cada uno con costo  $f(n/b)$ . (Es conveniente pensar en a como siendo

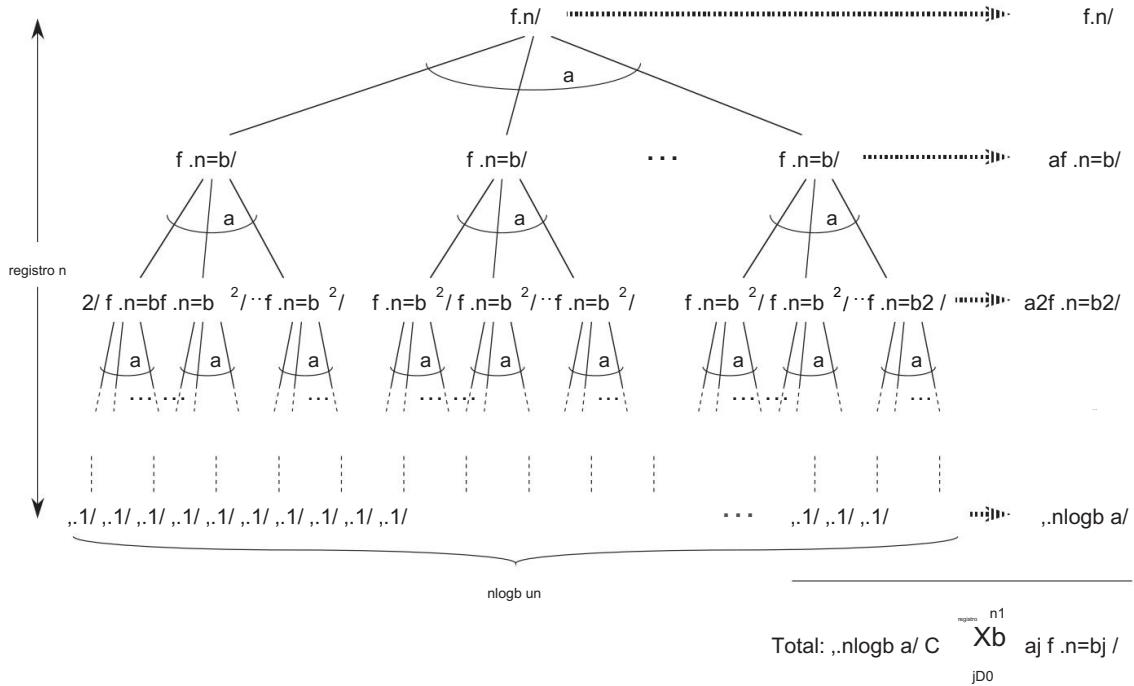


Figura 4.7 El árbol de recurrencia generado por  $T(n) = aT(n/b) + f(n)$ . El árbol es un árbol a-ario completo con hojas  $n \log b$  y altura  $\log b$ . El costo de los nudos en cada profundidad se muestra a la derecha y su suma se da en la ecuación (4.21).

un número entero, especialmente cuando se visualiza el árbol de recurrencia, pero las matemáticas no lo requieren.) Cada uno de estos hijos tiene un hijo, haciendo  $a^2$  nodos en la profundidad 2, y cada uno de los hijos tiene un costo  $f(n/b^2)$ . En general, hay  $a^j$  nodos en y cada hoja está uno tiene un costo  $f(n/b^j)$ . El costo de cada hoja es  $T(1) / D$ , y profundidad  $j$ , cada a profundidad  $\log b$ , ya que  $n = b^{\log b}$ . Hay  $a \log b$  hojas en el árbol.

Podemos obtener la ecuación (4.21) sumando los costos de los nodos en cada profundidad del árbol, como se muestra en la figura. El costo de todos los nodos internos en la profundidad  $j$  es  $af(n/b^j)$ , por lo que el costo total de todos los nodos internos es

$$\sum_{j=0}^{\log b} af(n/b^j)$$

En el algoritmo subyacente de divide y vencerás, esta suma representa los costos de dividir los problemas en subproblemas y luego recombinarlos. El

el costo de todas las hojas, que es el costo de hacer todos los subproblemas  $n \log b$  a de tamaño 1, es  $\dots n \log b a /$ .

En términos del árbol de recursión, los tres casos del teorema maestro corresponden a casos en los que el costo total del árbol está (1) dominado por los costos en las hojas, (2) distribuido uniformemente entre los niveles del árbol, o (3) dominado por el costo de la raíz.

La suma en la ecuación (4.21) describe el costo de los pasos de dividir y combinar en el algoritmo subyacente de divide y vencerás. El siguiente lema proporciona límites asintóticos sobre el crecimiento de la sumatoria.

Lema 4.3

Sean  $a$  y  $b > 1$  constantes, y sea  $f$  una función no negativa definida sobre potencias exactas de  $b$ . Una función  $q_n$  definida sobre potencias exactas de  $b$  por

$$gn/D \quad \begin{matrix} \text{registro} & n1 \\ Xb & aj f .n=bj / \\ iD0 & \end{matrix} \quad (4.22)$$

tiene los siguientes límites asintóticos para potencias exactas de b

1. Si  $f(n) \geq Cn \log n$  para alguna constante  $C > 0$ , entonces  $g(n) \leq Dn \log n$ .
  2. Si  $f(n) \leq cn \log n$ , entonces  $g(n) \geq \frac{c}{2}n \log n$ .
  3. Si  $af(n) + bg(n) \leq cf(n)$  para alguna constante  $c < 1$  y para todo  $n$  suficientemente grande, entonces  $g(n) \leq Dn$ .

Prueba Para el caso 1, tenemos  $f \cdot n / D O \cdot \log b / a$ , lo que implica que  $f \cdot n = b / D O \cdot \log b / a$ . Sustituyendo en la ecuación (4.22) se obtiene

g/ hacer <b>Xb</b> aj   — ¡D0	registro   n1 <small>note</small> <small>mamada</small>	Iniciar sesión ! :	(4.23)
-------------------------------------	---	--------------------	--------

Limitamos la suma dentro de la notación O factorizando términos y simplificando, lo que deja una serie geométrica creciente:

<b>Xb</b> <b>aj</b> <b>jD0</b>	<small>norte</small> <small>memoria</small>	<small>Iniciar sesión</small>	<b>D nlogb a</b>  <b>D nlogb a</b>  <b>D nlogb a</b>	<small>registro</small> <small>jD0</small> <small>jD0</small> <small>b registro n</small>	<b>Xb</b>  <b>Xb</b>  <b>.b /</b>	<small>abdominales</small>  <small>blogb un</small>  <small>1</small>
--------------------------------------	--	-------------------------------	--	--	---	---

$$\frac{D \cdot n \log b}{D \cdot O(n)} = \frac{n^{\frac{1}{1}}}{\text{segundo } 1}$$

Como  $b$  y  $O$  son constantes, podemos reescribir la última expresión como  $n \log b / D \cdot O(n)$ . Sustituyendo esta expresión por la suma en la ecuación (4.23) se obtiene

$gn / D \cdot O(n) \log b / D \cdot O(n)$  ;

probando así el caso 1.

Como el caso 2 asume que  $f(n) / D \cdot n \log b / D \cdot O(n)$ , tenemos que  $f(n) = bj / D \cdot n = bj / \log b / D \cdot O(n)$ .

Sustituyendo en la ecuación (4.22) se obtiene

$$\begin{array}{c} \text{registro } n \\ \text{Xb } aj \\ jD0 \end{array} \quad \begin{array}{c} \text{norte} \\ \text{mamada} \\ \text{Iniciar sesión ! :} \end{array} \quad (4.24)$$

Acotamos la suma dentro de la notación, como en el caso 1, pero esta vez no obtenemos una serie geométrica. En cambio, descubrimos que todos los términos de la suma son iguales:

$$\begin{array}{c} \text{registro } n \\ \text{Xb } aj \\ jD0 \end{array} \quad \begin{array}{c} \text{norte} \\ \text{mamada} \\ \text{Iniciar sesión ! :} \end{array} \quad \begin{array}{c} \text{registro } n \\ D \cdot n \log b \\ jD0 \end{array} \quad \begin{array}{c} \text{registro } n \\ \text{Xb } a \\ jD0 \end{array} \quad \begin{array}{c} \text{norte} \\ \text{blogb } un \\ \text{registro } n \\ \text{Xb } 1 \\ jD0 \end{array} \quad \begin{array}{c} \text{a} \\ \text{blogb } un \\ \text{registro } n \\ \text{Xb } 1 \\ jD0 \end{array}$$

$$D \cdot n \log b \cdot \log b / D \cdot n \log b$$

Sustituyendo esta expresión por la suma en la ecuación (4.24) se obtiene

$gn / D \cdot n \log b \cdot \log b / D \cdot n \log b$

$a \lg n /$  ;

prueba del caso 2.

Probamos el caso 3 de manera similar. Como  $f(n)$  aparece en la definición (4.22) de  $gn$  y todos los términos de  $gn$  son no negativos, podemos concluir que  $gn / D \cdot f(n)$  para potencias exactas de  $b$ . Suponemos en el enunciado del lema que  $af(n) = b^f$  para alguna constante  $c < 1$  y todas  $n$  suficientemente grandes. Reescribimos esta suposición como  $f(n) = bj / .c = a/f(n)$  e iteramos  $j$  veces, dando como resultado  $f(n) = bj / .c = a/f(n)^j$  o, de manera equivalente,  $ajf(n) = bj / .c^j = a/f(n)^j$ , donde asumimos que los valores que iteramos son lo suficientemente grandes. Dado que el último y más pequeño de tales valores es  $n = bj$ , es suficiente suponer que  $n = bj$  es suficientemente grande.

Sustituyendo en la ecuación (4.22) y simplificando se obtiene una serie geométrica, pero a diferencia de la serie del caso 1, esta tiene términos decrecientes. Usamos un término  $O(1)$  para

capturar los términos que no están cubiertos por nuestra suposición de que  $n$  es lo suficientemente grande:

$$\begin{aligned}
 & \text{gn/D} \quad Xb \underset{\substack{\text{registro} \\ jD0}}{ajf} .n=bj / \\
 & \quad Xb \underset{\substack{\text{registro} \\ jD0}}{cjf} .n/ C O.1/ \\
 & f.n/X1 \quad cj C O.1/ \\
 & \quad jD0 \\
 & D f .n/ \quad \frac{1}{1 c} \quad CO.1/ \\
 & D \text{ de } .n// ;
 \end{aligned}$$

ya que  $c$  es una constante. Por tanto, podemos concluir que  $gn/D ,f .n//$  para potencias exactas de  $b$ . Con el caso 3 probado, la demostración del lema está completa. ■

Ahora podemos probar una versión del teorema maestro para el caso en el que  $n$  es una potencia exacta de  $b$ .

#### Lema 4.4

Sean  $a$  y  $b > 1$  constantes, y sea  $f .n/$  una función no negativa definida sobre potencias exactas de  $b$ . Defina  $T .n/$  en potencias exactas de  $b$  por la recursión

$$\begin{aligned}
 & \text{si } n \neq 1 : \\
 & T .n/ = D ( ,a^T .n=b/ C f .n/ \text{ si } n \neq b) \\
 & \quad \vdots
 \end{aligned}$$

donde  $i$  es un entero positivo. Entonces  $T .n/$  tiene los siguientes límites asintóticos para potencias exactas de  $b$ :

1. Si  $f .n/ \leq D O.n \log_b a/$  para alguna constante  $> 0$ , entonces  $T .n/ \leq D ,n \log_b a/$ .
2. Si  $f .n/ \geq D ,n \log_b a/$ , entonces  $T .n/ \geq D ,n \log_b a \lg n/$ .
3. Si  $f .n/ \leq D .n \log_b a/C/$  para alguna constante  $> 0$ , y si  $af .n=b/ \leq cf .n/$  para alguna constante  $c < 1$  y todas  $n$  suficientemente grandes, entonces  $T .n/ \leq D ,f .n//$ .

Prueba Usamos los límites en el Lema 4.3 para evaluar la sumatoria (4.21) del Lema 4.2. Para el caso 1, tenemos

$$\begin{aligned}
 & T .n/ \leq D ,n \log_b a/ C O.n \log_b a/D \\
 & \quad ,n \log_b a/ ;
 \end{aligned}$$

y para el caso 2,

$$\begin{aligned} T .n/ D ,.nlogb a/ C ,.nlogb a \lg n/ D ,.nlogb \\ a \lg n/ : \end{aligned}$$

Para el caso 3,

$$T .n/ D ,.nlogb a/ C ,f .n// D ,f .n// ;$$

porque  $f .n/ D .nlogb aC/$ . ■

#### 4.6.2 Pisos y techos

Para completar la demostración del teorema maestro, ahora debemos extender nuestro análisis a la situación en la que aparecen pisos y techos en la recurrencia maestra, de modo que la recurrencia esté definida para todos los números enteros, no solo para las potencias exactas de b. Obtención un límite inferior en

$$T .n/ D aT .dn=be/ C f .n/ \quad (4.25)$$

y un límite superior en

$$T .n/ D aT .bn=bc/ C f .n/ \quad (4.26)$$

es una rutina, ya que podemos superar el límite  $dn=be$   $n=b$  en el primer caso para obtener el resultado deseado, y podemos superar el límite  $bn=bc$   $n=b$  en el segundo caso. Usamos la misma técnica para el límite inferior de la recurrencia (4.26) que para el límite superior de la recurrencia (4.25), por lo que presentaremos solo este último límite.

Modificamos el árbol de recurrencia de la figura 4.7 para producir el árbol de recurrencia de la figura 4.8. A medida que descendemos en el árbol de recursividad, obtenemos una secuencia de invocaciones recursivas sobre los argumentos

```
note : ;
dn=ser;
ddn=ser =ser ;
dddn=ser =ser =ser ;

```

⋮

Denotemos el j-ésimo elemento en la secuencia por  $n_j$ , donde

$$\begin{aligned} & \text{si } j \leq 0 \quad : \\ n_j D (n - dnj1=ser \text{ si } j > 0: \quad (4.27) \end{aligned}$$

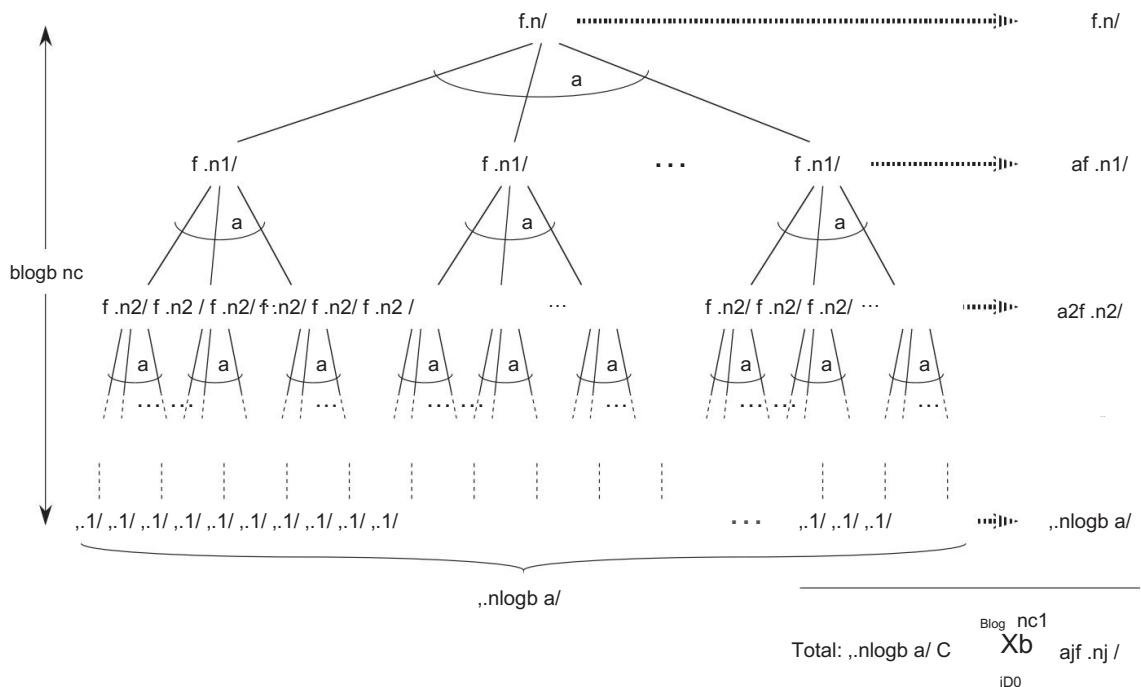
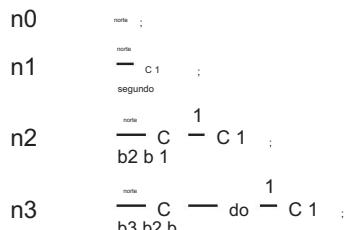
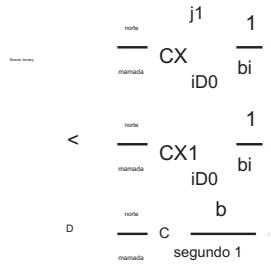


Figura 4.8 El árbol de recurrencia generado por  $T(n) = 2T(n/2) + O(n)$ . El argumento recursivo  $n_j$  viene dado por la ecuación (4.27).

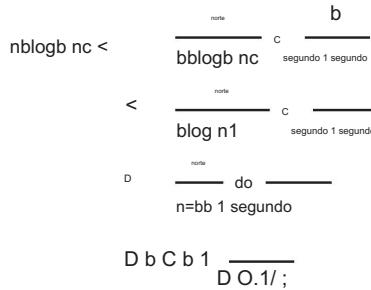
Nuestro primer objetivo es determinar la profundidad  $k$  tal que  $nk$  sea una constante. Usando la desigualdad dxe  $x C 1$ , obtenemos



En general, tenemos



Dejando  $j D \log b n c$ , obtenemos



y así vemos que en profundidad  $\log b n c$ , el tamaño del problema es como mucho una constante.

De la Figura 4.8, vemos que

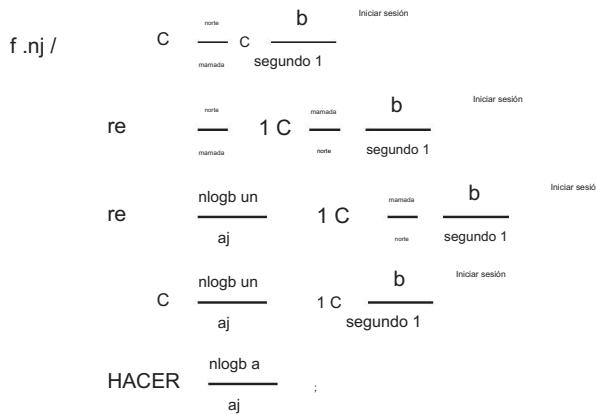
$$\begin{aligned} T .n/ D ,n \log b a/ C & \quad \text{Blog } nc1 \\ & \quad Xb \quad aj f .nj / ; \\ & \quad jD0 \end{aligned} \tag{4.28}$$

que es muy similar a la ecuación (4.21), excepto que  $n$  es un número entero arbitrario y no está restringido a ser una potencia exacta de  $b$ .

Ahora podemos evaluar la suma

$$\begin{aligned} gn/D & \quad \text{Blog } nc1 \\ & \quad Xb \quad aj f .nj / \\ & \quad jD0 \end{aligned} \tag{4.29}$$

de la ecuación (4.28) de manera análoga a la demostración del Lema 4.3. Comenzando con el caso 3, si  $af .dn = be/ cf .n/$  para  $n > bCb = b1/$ , donde  $c < 1$  es una constante, entonces se sigue que  $ajf .nj / < cj f .n/$ . Por lo tanto, podemos evaluar la suma en la ecuación (4.29) tal como en el Lema 4.3. Para el caso 2, tenemos  $f .n/ D ,n \log b a/$ . Si podemos mostrar que  $f .nj / D O.n \log b a = aj / D O..n = bj / \log b a/$ , entonces la prueba para el caso 2 del Lema 4.3 se cumplirá. Observe que  $j \log b n c$  implica  $bj = n 1$ . La cota  $f .n/ D O.n \log b a/$  implica que existe una constante  $c > 0$  tal que para todo  $nj$  suficientemente grande,



ya que  $c \cdot 1 \cdot C \cdot b = b \cdot 1 // \logb a$  es una constante. Así, hemos probado el caso 2. La demostración del caso 1 es casi idéntica. La clave es probar el límite  $f(n) / D \cdot O(n \logb a)$ , que es similar a la prueba correspondiente del caso 2, aunque el álgebra es más compleja.

Ahora hemos demostrado los límites superiores en el teorema maestro para todos los números enteros  $n$ . La demostración de las cotas inferiores es similar.

### Ejercicios

#### 4.6-1 ?

Dé una expresión simple y exacta para  $n_j$  en la ecuación (4.27) para el caso en que  $b$  sea un entero positivo en lugar de un número real arbitrario.

#### 4.6-2 ?

Demuestre que si  $f(n) / D \cdot n \logb a \lgk n$ , donde  $k > 0$ , entonces la recurrencia maestra tiene solución  $T(n) / D \cdot n \logb a \lgk C_1 n$ . Para simplificar, limite su análisis a potencias exactas de  $b$ .

#### 4.6-3 ?

Muestre que el caso 3 del teorema maestro está exagerado, en el sentido de que la condición de regularidad  $a_f n = b / c f(n)$  para alguna constante  $c < 1$  implica que existe una constante  $> 0$  tal que  $f(n) / D \cdot n \logb a c$ .

## Problemas

### 4-1 Ejemplos de recurrencia Dé

límites asintóticos superior e inferior para  $T(n)$  en cada una de las siguientes recurrencias. Suponga que  $T(n)$  es constante para  $n \geq 2$ . Haga sus límites lo más ajustados posible y justifique sus respuestas.

- a.  $T(n) \leq 2T(n/2) + Cn^4$ .
- b.  $T(n) \leq DT(7n/10) + Cn$ .
- c.  $T(n) \leq 16T(n/4) + Cn^2$ .
- d.  $T(n) \leq 7T(n/3) + Cn^2$ .
- e.  $T(n) \leq 7T(n/2) + Cn^2$ .
- F.  $T(n) \leq 2T(n/4) + Cpn$ .
- gramo.  $T(n) \leq DT(n/2) + Cn^2$ .

—

### 4-2 Costos de paso de parámetros A

lo largo de este libro, asumimos que el paso de parámetros durante las llamadas a procedimientos lleva un tiempo constante, incluso si se pasa una matriz de  $N$  elementos. Esta suposición es válida en la mayoría de los sistemas porque se pasa un puntero a la matriz, no la matriz en sí.

Este problema examina las implicaciones de tres estrategias de paso de parámetros:

1. Una matriz se pasa por puntero. Tiempo  $D_{\text{copy}} = O(N)$ .
  2. Una matriz se pasa copiando. Tiempo  $D_{\text{copy}} = O(N^2)$ , donde  $N$  es el tamaño de la matriz.
  3. Se pasa una matriz copiando solo el subrango al que podría acceder el procedimiento llamado. Tiempo  $D_{\text{copy}} = O(qN)$  si se pasa el subarreglo  $A[1:N] : q$ .
- a. Considere el algoritmo de búsqueda binaria recursiva para encontrar un número en una matriz ordenada (vea el ejercicio 2.3-5). Proporcione recurrencias para los tiempos de ejecución de búsqueda binaria en el peor de los casos cuando se pasan matrices utilizando cada uno de los tres métodos anteriores, y proporcione buenos límites superiores en las soluciones de las recurrencias. Sea  $N$  el tamaño del problema original y  $n$  el tamaño de un subproblema.
  - b. Rehaga la parte (a) para el algoritmo MERGE-SORT de la Sección 2.3.1.

## 4-3 Más ejemplos de recurrencia Dé

límites asintóticos superior e inferior para  $T(n)$  en cada una de las siguientes recurrencias.

Suponga que  $T(n)$  es constante para  $n$  suficientemente pequeño. Haga sus límites lo más ajustados posible y justifique sus respuestas.

- a.  $T(n) \leq 4T(n/3) + C n \lg n$ .
- b.  $T(n) \leq 3T(n/3) + C n = \lg n$ .
- c.  $T(n) \leq 4T(n/2) + C n^2 p_n$ .
- d.  $T(n) \leq 3T(n/3) + C n^2$ .
- e.  $T(n) \leq 2T(n/2) + C n = \lg n$ .
- f.  $T(n) \leq DT(n/2) + CT(n/4) + CT(n/8) + C n$ .
- gramo.  $T(n) \leq DT(n/1) + C 1 = n$ .
- g.  $T(n) \leq DT(n/1) + C \lg n$ .
- i.  $T(n) \leq DT(n/2) + C 1 = \lg n$ .
- j.  $T(n) \leq D p_n T(n/p_n) + C n$ .

## 4-4 números de Fibonacci

Este problema desarrolla propiedades de los números de Fibonacci, que se definen por recurrencia (3.22). Usaremos la técnica de generación de funciones para resolver la recurrencia de Fibonacci. Defina la función generadora (o serie formal de potencias)  $F$  como

$$F := \sum_{i=0}^{\infty} F_i x^i = F_0 + F_1 x + F_2 x^2 + F_3 x^3 + F_4 x^4 + \dots$$

$$= 1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + 13x^6 + 21x^7 + 34x^8 + \dots$$

donde  $F_i$  es el  $i$ -ésimo número de Fibonacci.

- a. Demuestre que  $F := 1 + F + x^2 F$ .

b. Muestra esa

$$\begin{array}{r} F \cdot / D \\ \hline 1 \cdot \cdot 2 \cdot \\ \hline \begin{array}{c} D \\ .1 \end{array} \quad \begin{array}{c} \cdot / .1 \\ 1 \end{array} \quad \begin{array}{c} \cdot / \\ 1 \end{array} \\ \hline \begin{array}{c} D \\ 1 \\ \hline p5 \end{array} \quad \begin{array}{c} 1 \\ \hline 1 \end{array} \quad \begin{array}{c} 1 \\ \hline 1 \cdot y; \end{array} \end{array}$$

dónde

$$\begin{array}{r} D \\ \hline 1 \cdot p5 \\ 2 \end{array} \quad D \ 1:61803 \dots$$

y

$$\begin{array}{r} D \\ \hline 1 \cdot p5 \\ 2 \end{array} \quad D \ 0:61803 \dots$$

C. Muestra esa

$$F \cdot / D X_1 \frac{1}{iD_0} \cdot i \quad y_0 / i :$$

d. Use la parte (c) para demostrar que  $F_i D^i = p5$  para  $i > 0$ , redondeado al entero más próximo.  
 (Sugerencia: Observe que  $\cdot y \cdot < 1$ .)

#### 4-5 Prueba de chips

El profesor Diógenes tiene  $n$  chips de circuitos integrados supuestamente idénticos que, en principio, son capaces de probarse entre sí. La plantilla de prueba del profesor acomoda dos chips a la vez. Cuando se carga la plantilla, cada chip prueba al otro e informa si es bueno o malo. Un buen chip siempre informa con precisión si el otro chip es bueno o malo, pero el profesor no puede confiar en la respuesta de un mal chip. Por lo tanto, los cuatro posibles resultados de una prueba son los siguientes:

El chip A dice El chip B dice Conclusión B es

bueno A es bueno ambos son buenos, o ambos son malos B es  
 bueno A es malo al menos uno es malo B es malo A  
 es bueno al menos uno es malo B es malo A es malo  
 al menos uno es malo

a. Muestre que si más de  $n=2$  fichas son malas, el profesor no necesariamente puede determinar qué fichas son buenas utilizando ninguna estrategia basada en este tipo de prueba por pares. Suponga que las fichas malas pueden conspirar para engañar al profesor.

- b. Considere el problema de encontrar una sola ficha buena entre  $n$  fichas, suponiendo que más de  $n=2$  de las fichas son buenas. Demuestre que  $bn=2c$  las pruebas por pares son suficientes para reducir el problema a uno de casi la mitad del tamaño.
- C. Muestre que las fichas buenas se pueden identificar con pruebas  $,n/$  por pares, suponiendo que más de  $n=2$  de las fichas son buenas. Dar y resolver la recurrencia que describe el número de pruebas.

#### 4-6 arreglos Monge

Un arreglo  $m \times n$   $A$  de números reales es un arreglo de Monge si para todos los  $i$ ,  $j$ ,  $k$  y  $l$  tales que  $1 \leq i < m$  y  $1 \leq j < n$ , tenemos

$$A[i][j] \leq A[i][k] \leq A[i][l] \quad A[i][j] \leq A[k][j] \leq A[l][j]$$

En otras palabras, cada vez que elegimos dos filas y dos columnas de una matriz de Monge y consideramos los cuatro elementos en las intersecciones de las filas y las columnas, la suma de los elementos superior izquierdo e inferior derecho es menor o igual que el suma de los elementos inferior izquierdo y superior derecho. Por ejemplo, la siguiente matriz es Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
33	19	21	6	75
53	34			

- a. Demostrar que un arreglo es Monge si y solo si para todo  $i \in D[1; 2; \dots; m]$  y  $j \in D[1; 2; \dots; n]$ , tenemos

$$A[i][j] \leq A[i][C[1; j]] \leq A[i][C[1; j]] \leq A[C[i][1; j]] \leq A[C[i][1; j]]$$

(Sugerencia: para la parte "si", use la inducción por separado en filas y columnas).

- b. La siguiente matriz no es Monge. Cambiar un elemento para hacerlo Monge. (Sugerencia: use la parte (a).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21		
15	8		

C. Sea  $f.i$  / el índice de la columna que contiene el elemento mínimo más a la izquierda de la fila  $i$ .

Demuestre que  $f.1/f.2/f.m$  para cualquier arreglo de  $m n$  Monge.

d. Aquí hay una descripción de un algoritmo divide y vencerás que calcula el elemento mínimo más a la izquierda en cada fila de una matriz A de  $m n$  Monge:

Construya una submatriz  $A_0$  de A que consista en las filas pares de A.

Determine recursivamente el mínimo más a la izquierda para cada fila de  $A_0$ . Luego calcule el mínimo más a la izquierda en las filas impares de A.

Explique cómo calcular el mínimo más a la izquierda en las filas impares de A (dado que se conoce el mínimo más a la izquierda de las filas pares) en tiempo  $O(m C n)$ .

mi. Escriba la recurrencia que describe el tiempo de ejecución del algoritmo descrito en la parte (d).

Demuestre que su solución es  $O(m C n \log m)$ .

## Notas del capítulo

Divide y vencerás como técnica para diseñar algoritmos se remonta al menos a 1962 en un artículo de Karatsuba y Ofman [194]. Sin embargo, podría haber sido usado mucho antes; según Heideman, Johnson y Burrus [163], CF Gauss ideó el primer algoritmo de transformada rápida de Fourier en 1805, y la formulación de Gauss divide el problema en subproblemas más pequeños cuyas soluciones se combinan.

El problema del subarreglo máximo de la Sección 4.1 es una variación menor de un problema estudiado por Bentley [43, Capítulo 7].

El algoritmo de Strassen [325] causó mucho entusiasmo cuando se publicó en 1969. Antes de eso, pocos imaginaban la posibilidad de un algoritmo asintóticamente más rápido que el procedimiento básico SQUARE-MATRIX-MULTIPLY . El límite superior asintótico para la multiplicación de matrices se ha mejorado desde entonces. El algoritmo más eficiente asintóticamente para multiplicar matrices  $n n$  hasta la fecha, debido a Coppersmith y Winograd [78], tiene un tiempo de ejecución de  $O(n^{2.376})$ . El mejor límite inferior conocido es simplemente el límite  $n^2$  / obvio (obvio porque debemos completar  $n^2$  elementos de la matriz del producto).

Desde un punto de vista práctico, el algoritmo de Strassen a menudo no es el método de elección para la multiplicación de matrices, por cuatro razones:

1. El factor constante oculto en el tiempo de ejecución  $\cdot n^{lg 7}$  / del algoritmo de Strassen es mayor que el factor constante en el procedimiento  $\cdot n^3$ -time SQUARE-MATRIX MULTIPLY .
2. Cuando las matrices son dispersas, los métodos adaptados para matrices dispersas son más rápidos.

3. El algoritmo de Strassen no es tan estable numéricamente como SQUARE-MATRIX

MULTIPLY. En otras palabras, debido a la precisión limitada de la aritmética computarizada en valores no enteros, se acumulan errores más grandes en el algoritmo de Strassen que en SQUARE-MATRIX-MULTIPLY.

4. Las submatrices formadas en los niveles de recursividad consumen espacio.

Las dos últimas razones se mitigaron alrededor de 1990. Higham [167] demostró que se había exagerado la diferencia en la estabilidad numérica; aunque el algoritmo de Strassen es demasiado inestable numéricamente para algunas aplicaciones, está dentro de los límites aceptables para otras. Bailey, Lee y Simon [32] discuten técnicas para reducir los requisitos de memoria para el algoritmo de Strassen.

En la práctica, las implementaciones rápidas de multiplicación de matrices para matrices densas usan el algoritmo de Strassen para tamaños de matriz por encima de un "punto de cruce" y cambian a un método más simple una vez que el tamaño del subproblema se reduce por debajo del punto de cruce. El valor exacto del punto de cruce depende en gran medida del sistema. Los análisis que cuentan las operaciones pero ignoran los efectos de los cachés y la canalización han producido puntos de cruce tan bajos como  $n D 8$  (por Higham [167]) o  $n D 12$  (por Huss-Lederman et al. [186]). D'Alberto y Nicolau [81] desarrollaron un esquema adaptativo que determina el punto de cruce mediante la evaluación comparativa cuando se instala su paquete de software. Encontraron puntos de cruce en varios sistemas que van desde  $n D 400$  a  $n D 2150$ , y no pudieron encontrar un punto de cruce en un par de sistemas.

Las recurrencias fueron estudiadas ya en 1202 por L. Fibonacci, por quien se nombran los números de Fibonacci. A. De Moivre introdujo el método de generación de funciones (vea el problema 4-4) para resolver recurrencias. El método maestro está adaptado de Bentley, Haken y Saxe [44], que proporciona el método extendido justificado por el ejercicio 4.6-2. Knuth [209] y Liu [237] muestran cómo resolver recurrencias lineales utilizando el método de generación de funciones. Purdom y Brown [287] y Graham, Knuth y Patashnik [152] contienen discusiones extensas sobre resolución de recurrencia.

Varios investigadores, incluidos Akra y Bazzi [13], Roura [299], Verma [346] y Yap [360], han proporcionado métodos para resolver recurrencias más generales de divide y vencerás que las que se resuelven con el método maestro. Aquí describimos el resultado de Akra y Bazzi, modificado por Leighton [228]. El método Akra-Bazzi funciona para recurrencias de la forma

$$\text{si } 1 \leq x < 0 ; a_i T . b_i x^i \quad (4.30)$$

$$T . x / D ( , 1 / \frac{1}{x} )^{D-1} C f . x / \text{ si } x > x_0 ;$$

dónde

$x_0$  es un número real,

$x_0$  es una constante tal que  $x_0 = b_i$  y  $x_0 = -b_i$  para  $i = 1, 2, \dots, k$ ,  $a_i$  es una constante positiva para  $i = 1, 2, \dots, k$ ,

$b_i$  es una constante en el rango  $0 < b_i < 1$  para  $i \in D_1; 2; \dots; k, k \in$

es una constante entera, y

$f(x)$  es una función no negativa que satisface la condición de crecimiento polinomial: existen constantes positivas  $c_1$  y  $c_2$  tales que para todo  $x \geq 1$ , para  $i \in D_1; 2; \dots; k$ , y para todo  $u$  tal que  $b_i u \leq x$ , tenemos  $c_1 f(u) \leq f(x) \leq c_2 f(u)$ . (Si  $\limsup_{x \rightarrow \infty} f(x)/x^y$  existe, entonces  $f(x)$  satisface la condición de crecimiento del polinomio. Por ejemplo,  $f(x) = x^{\alpha}$  satisface esta condición para cualquier constante real  $\alpha > 0$ .)

Aunque el método maestro no se aplica a una recurrencia como  $T(n) = DT.bn + C$  ( $D, b, C$  constantes), el método de Akra-Bazzi sí lo hace. Para resolver la recurrencia (4.30), primero

encontramos el único número real  $p$  tal que  $\sum_{i=1}^k b_i^p = 1$ .

(Tal  $p$  siempre existe.) La solución a la recurrencia es entonces

$$T(n) = \int_1^n \frac{f(u)}{u^{p+1}} du$$

El método de Akra-Bazzi puede ser algo difícil de usar, pero sirve para resolver recurrencias que modelan la división del problema en subproblemas de tamaño sustancialmente desigual. El método maestro es más simple de usar, pero solo se aplica cuando los tamaños de los subproblemas son iguales.

---

## 5

# Análisis Probabilístico y Aleatorizado Algoritmos

Este capítulo introduce el análisis probabilístico y los algoritmos aleatorios. Si no está familiarizado con los conceptos básicos de la teoría de la probabilidad, debe leer el Apéndice C, que revisa este material. Revisaremos el análisis probabilístico y los algoritmos aleatorios varias veces a lo largo de este libro.

---

## 5.1 El problema de la contratación

Suponga que necesita contratar a un nuevo asistente de oficina. Sus intentos anteriores de contratación no han tenido éxito y decide utilizar una agencia de empleo. La agencia de empleo le envía un candidato cada día. Entrevistas a esa persona y luego decides contratarla o no. Debe pagar a la agencia de empleo una pequeña tarifa para entrevistar a un solicitante. Sin embargo, contratar a un solicitante es más costoso, ya que debe despedir a su asistente de oficina actual y pagar una tarifa de contratación sustancial a la agencia de empleo. Usted se compromete a contar en todo momento con la mejor persona posible para el puesto. Por lo tanto, decide que, después de entrevistar a cada solicitante, si ese solicitante está mejor calificado que el asistente de oficina actual, despedirá al asistente de oficina actual y contratará al nuevo solicitante. Está dispuesto a pagar el precio resultante de esta estrategia, pero desea estimar cuál será ese precio.

El procedimiento **CONTRATAR-ASISTENTE**, que se presenta a continuación, expresa esta estrategia de contratación en pseudocódigo. Supone que los candidatos para el trabajo de asistente de oficina están numerados del 1 al  $n$ . El procedimiento asume que usted puede, después de entrevistar al candidato  $i$ , determinar si el candidato  $i$  es el mejor candidato que ha visto hasta ahora. Para inicializar, el procedimiento crea un candidato ficticio, con el número 0, que está menos calificado que cada uno de los demás candidatos.

### HIRE-ASSISTANT.n/ 1

mejor D 0 // el candidato 0 es un candidato ficticio menos calificado 2 para i D 1 a n 3 entrevistar al candidato i 4 si el candidato i es mejor que el candidato mejor 5 mejor D i 6 contratar al candidato i

El modelo de costos para este problema difiere del modelo descrito en el Capítulo 2. No nos enfocamos en el tiempo de ejecución de HIRE-ASSISTANT, sino en los costos incurridos por la entrevista y la contratación. En la superficie, analizar el costo de este algoritmo puede parecer muy diferente de analizar el tiempo de ejecución de, por ejemplo, la ordenación por fusión. Las técnicas analíticas utilizadas, sin embargo, son idénticas ya sea que analicemos el costo o el tiempo de ejecución. En cualquier caso, estamos contando el número de veces que se ejecutan ciertas operaciones básicas.

Entrevistar tiene un costo bajo, digamos  $c_i$ , mientras que contratar es costoso, cuesta  $ch_m$ . Sea  $m$  el número de personas contratadas, el coste total asociado a este algoritmo es  $O(cin C chm)$ . No importa cuántas personas contratemos, siempre entrevistamos a  $n$  candidatos y, por lo tanto, siempre incurrimos en el costo  $cin$  asociado con la entrevista. Por lo tanto, nos concentraremos en analizar  $chm$ , el costo de contratación. Esta cantidad varía con cada ejecución del algoritmo.

Este escenario sirve como modelo para un paradigma computacional común. A menudo necesitamos encontrar el valor máximo o mínimo en una secuencia examinando cada elemento de la secuencia y manteniendo un "ganador" actual. El problema de contratación modela la frecuencia con la que actualizamos nuestra noción de qué elemento está ganando actualmente.

#### Análisis del peor de los casos

En el peor de los casos, contratamos a todos los candidatos que entrevistamos. Esta situación se da si los candidatos vienen en estricto orden creciente de calidad, en cuyo caso contratamos  $n$  veces, por un costo total de contratación de  $O(chn)$ .

Por supuesto, los candidatos no siempre vienen en orden creciente de calidad. De hecho, no tenemos idea del orden en que llegan, ni tenemos ningún control sobre este orden. Por lo tanto, es natural preguntarse qué esperamos que suceda en un caso típico o promedio.

#### Análisis probabilístico

El análisis probabilístico es el uso de la probabilidad en el análisis de problemas. Más comúnmente, usamos el análisis probabilístico para analizar el tiempo de ejecución de un algoritmo. A veces lo usamos para analizar otras cantidades, como el coste de contratación

en trámite CONTRATAR-ASISTENTE. Para realizar un análisis probabilístico, debemos usar el conocimiento o hacer suposiciones sobre la distribución de las entradas.

Luego analizamos nuestro algoritmo, calculando un tiempo de ejecución de caso promedio, donde tomamos el promedio sobre la distribución de las posibles entradas. Por lo tanto, estamos, en efecto, promediando el tiempo de ejecución de todas las entradas posibles. Al informar sobre un tiempo de ejecución de este tipo, nos referiremos a él como el tiempo de ejecución del caso promedio.

Debemos ser muy cuidadosos al decidir sobre la distribución de insumos. Para algunos problemas, podemos suponer razonablemente algo sobre el conjunto de todas las entradas posibles, y luego podemos usar el análisis probabilístico como técnica para diseñar un algoritmo eficiente y como un medio para obtener información sobre un problema. Para otros problemas, no podemos describir una distribución de entrada razonable y, en estos casos, no podemos usar el análisis probabilístico.

Para el problema de contratación, podemos suponer que los solicitantes vienen en orden aleatorio. ¿Qué significa eso para este problema? Suponemos que podemos comparar dos candidatos cualesquiera y decidir cuál está mejor calificado; es decir, hay un orden total sobre los candidatos. (Vea el Apéndice B para la definición de un total o der.) Por lo tanto, podemos clasificar a cada candidato con un número único del 1 al n, usando rank.i / para indicar el rango del solicitante i, y adoptar la convención de que un mayor el rango corresponde a un solicitante mejor calificado. La lista ordenada hrank.1/; rango.2/; : : : ;rank.n/i es una permutación de la lista h1; 2; : : : ; ni. Decir que los solicitantes vienen en un orden aleatorio es equivalente a decir que esta lista de rangos tiene la misma probabilidad de ser cualquiera de las n! permutaciones de los números del 1 al n.

Alternativamente, decimos que los rangos forman una permutación aleatoria uniforme; es decir, cada una de las posibles permutaciones n! aparece con igual probabilidad.

La sección 5.2 contiene un análisis probabilístico del problema de contratación.

### Algoritmos aleatorizados

Para usar el análisis probabilístico, necesitamos saber algo sobre la distribución de las entradas. En muchos casos, sabemos muy poco sobre la distribución de entrada.

Incluso si sabemos algo sobre la distribución, es posible que no podamos modelar este conocimiento computacionalmente. Sin embargo, a menudo podemos usar la probabilidad y la aleatoriedad como una herramienta para el diseño y análisis de algoritmos, haciendo que el comportamiento de parte del algoritmo sea aleatorio.

En el problema de la contratación, puede parecer que los candidatos se nos presentan en un orden aleatorio, pero no tenemos forma de saber si realmente lo son o no.

Por lo tanto, para desarrollar un algoritmo aleatorio para el problema de contratación, debemos tener un mayor control sobre el orden en que entrevistamos a los candidatos. Por lo tanto, cambiaremos ligeramente el modelo. Decimos que la agencia de empleo tiene n candidatos, y nos envían una lista de los candidatos por adelantado. Cada día, elegimos, al azar, a qué candidato entrevistar. Aunque no sabemos nada sobre

los candidatos (además de sus nombres), hemos hecho un cambio significativo. En lugar de confiar en que los candidatos nos llegan en un orden aleatorio, hemos obtenido el control del proceso y hemos impuesto un orden aleatorio.

De manera más general, llamamos aleatorio a un algoritmo si su comportamiento está determinado no solo por su entrada sino también por los valores producidos por un generador de números aleatorios. Supondremos que tenemos a nuestra disposición un generador de números aleatorios RANDOM. Una llamada a RANDOM.a; b/ devuelve un entero entre a y b, inclusive, siendo cada uno de esos enteros igualmente probable. Por ejemplo, ALEATORIO.0; 1/ produce 0 con probabilidad 1=2, y produce 1 con probabilidad 1=2. Una llamada a RANDOM.3; 7/ devuelve 3, 4, 5, 6 o 7, cada uno con probabilidad 1=5. Cada entero devuelto por RANDOM es independiente de los enteros devueltos en llamadas anteriores.

Puede imaginarse RANDOM como tirar un dado .ba C de 1 cara para obtener su salida. (En la práctica, la mayoría de los entornos de programación ofrecen un generador de números pseudoaleatorios: un algoritmo determinista que devuelve números que "parecen" estadísticamente aleatorios).

Cuando analizamos el tiempo de ejecución de un algoritmo aleatorio, tomamos la expectativa del tiempo de ejecución sobre la distribución de valores devueltos por el generador de números aleatorios. Distinguimos estos algoritmos de aquellos en los que la entrada es aleatoria al referirnos al tiempo de ejecución de un algoritmo aleatorio como un tiempo de ejecución esperado. En general, analizamos el tiempo de ejecución del caso promedio cuando la distribución de probabilidad está sobre las entradas del algoritmo, y analizamos el tiempo de ejecución esperado cuando el propio algoritmo toma decisiones aleatorias.

## Ejercicios

### 5.1-1

Demuestre que la suposición de que siempre podemos determinar qué candidato es el mejor, en la línea 4 del procedimiento CONTRATAR-ASISTENTE, implica que conocemos un orden total en las filas de los candidatos.

### 5.1-2 ?

Describir una implementación del procedimiento RANDOM.a; b/ que solo hace llamadas a RANDOM.0; 1/. ¿Cuál es el tiempo de ejecución esperado de su procedimiento, en función de a y b?

### 5.1-3 ?

Suponga que desea generar 0 con probabilidad 1=2 y 1 con probabilidad 1=2. A su disposición hay un procedimiento BIASED-RANDOM, que da como resultado 0 o 1. Da como resultado 1 con alguna probabilidad p y 0 con probabilidad 1 p, donde 0<p<1, pero no sabe qué es p. Proporcione un algoritmo que use BIASED-RANDOM como una subrutina y devuelva una respuesta imparcial, devolviendo 0 con probabilidad 1 = 2

y 1 con probabilidad  $1=2$ . ¿Cuál es el tiempo de ejecución esperado de su algoritmo en función de  $p$ ?

## 5.2 Indicador de variables aleatorias

Para analizar muchos algoritmos, incluido el problema de la contratación, utilizamos variables aleatorias indicadoras. Las variables aleatorias indicadoras proporcionan un método conveniente para convertir entre probabilidades y expectativas. Supongamos que tenemos un espacio muestral  $S$  y un evento  $A$ . Entonces, la variable aleatoria indicadora  $I_{fAg}$  asociada con el evento  $A$  se define como

$$I_{fAg} = \begin{cases} 0 & \text{si } A \text{ no ocurre;} \\ 1 & \text{si } A \text{ ocurre}\end{cases} \quad (5.1)$$

Como ejemplo simple, determinemos el número esperado de caras que obtenemos al lanzar una moneda al aire. Nuestro espacio muestral es  $S = \{H, T\}$ , con  $\Pr(H) = \Pr(T) = 1/2$ . Entonces podemos definir una variable aleatoria indicadora  $X_H$ , con la moneda saliendo cara, que es el evento  $H$ . Esta variable cuenta el número de caras obtenidas en este lanzamiento, y es 1 si la moneda sale cara y 0 en caso contrario. Nosotros escribimos

$$X_H = I_{fHg}$$

$$D = \begin{cases} 0 & \text{si } H \text{ ocurre;} \\ 1 & \text{si } H \text{ no ocurre}\end{cases}$$

El número esperado de caras obtenidas en un lanzamiento de la moneda es simplemente el valor esperado de nuestra variable indicadora  $X_H$ :

$$E(X_H) = \Pr(H) \cdot 1 + \Pr(T) \cdot 0 = 1/2$$

$$\Pr(H) = 1/2$$

$$\Pr(T) = 1/2$$

Por lo tanto, el número esperado de caras obtenidas al lanzar una moneda justa es  $1/2$ . Como muestra el siguiente lema, el valor esperado de una variable aleatoria indicadora asociada con un evento  $A$  es igual a la probabilidad de que  $A$  ocurra.

Lema 5.1

Dado un espacio muestral  $S$  y un evento  $A$  en el espacio muestral  $S$ , sea  $X_A = I_{fAg}$ . Entonces  $E(X_A) = \Pr(A)$ .

Prueba Por la definición de una variable aleatoria indicadora de la ecuación (5.1) y la definición del valor esperado, tenemos

E EXA DE OEI fAg

$$D 1 \text{ Pr } fAg C 0 \text{ Pr } ^\circ A$$

$$DPr fAg;$$

donde  $\bar{A}$  denota SA, el complemento de A. ■

Aunque las variables aleatorias indicadoras pueden parecer engorrosas para una aplicación como contar el número esperado de caras al lanzar una sola moneda, son útiles para analizar situaciones en las que realizamos ensayos aleatorios repetidos. Por ejemplo, las variables aleatorias indicadoras nos brindan una forma sencilla de llegar al resultado de la ecuación (C.37). En esta ecuación, calculamos el número de caras en n lanzamientos de monedas considerando por separado la probabilidad de obtener 0 caras, 1 cara, 2 caras, etc. El método más simple propuesto en la ecuación (C.38) en su lugar utiliza implícitamente variables aleatorias indicadoras. Para hacer más explícito este argumento, dejamos que  $X_i$  sea la variable aleatoria indicadora asociada con el evento en el que el i-ésimo lanzamiento sale cara:  $X_i = 1$  si el i-ésimo lanzamiento da como resultado el evento Hg. Sea  $X$  la variable aleatoria que denota el número total de caras en los n lanzamientos de moneda, de modo que

$$XD = \sum_{i=1}^n X_i$$

Deseamos calcular el número esperado de caras, por lo que tomamos la expectativa de ambos lados de la ecuación anterior para obtener

$$E EXA = E \sum_{i=1}^n X_i$$

La ecuación anterior da la expectativa de la suma de n variables aleatorias indicadoras. Por el Lema 5.1, podemos calcular fácilmente la expectativa de cada una de las variables aleatorias. Por la ecuación (C.21) —linealidad de la expectativa— es fácil calcular la expectativa de la suma: es igual a la suma de las expectativas de las n variables aleatorias. La linealidad de la expectativa hace que el uso de variables aleatorias indicadoras sea una poderosa técnica analítica; se aplica incluso cuando existe dependencia entre las variables aleatorias. Ahora podemos calcular fácilmente el número esperado de ca

$$E \{EX\} = \sum_{i=1}^n p_i E[X_i]$$

$$X_n = \sum_{i=1}^n p_i X_i$$

$$X_n = \sum_{i=1}^n p_i X_i$$

re n=2 :

Por lo tanto, en comparación con el método utilizado en la ecuación (C.37), las variables aleatorias indicadoras simplifican enormemente el cálculo. Usaremos variables aleatorias indicadoras a lo largo de este libro.

#### Análisis del problema de contratación mediante variables aleatorias indicadoras

Volviendo al problema de la contratación, ahora deseamos calcular el número esperado de veces que contrataremos a un nuevo asistente de oficina. Para usar un análisis probabilístico, asumimos que los candidatos llegan en un orden aleatorio, como se discutió en la sección anterior. (Veremos en la Sección 5.3 cómo eliminar esta suposición.) Sea  $X$  la variable aleatoria cuyo valor es igual al número de veces que contratamos una nueva oficina como asistente. Entonces podríamos aplicar la definición de valor esperado de la ecuación (C.20) para obtener

$$E \{EX\} = \sum_{x=0}^n x \Pr \{X=x\}$$

pero este cálculo sería engorroso. En su lugar, utilizaremos variables aleatorias indicadoras para simplificar en gran medida el cálculo.

Para usar variables aleatorias indicadoras, en lugar de calcular  $E \{EX\}$  definiendo una variable asociada con el número de veces que contratamos a un nuevo asistente de oficina, definimos  $n$  variables relacionadas con si se contrata o no a cada candidato en particular. En particular, dejamos que  $X_i$  sea la variable aleatoria indicadora asociada al evento en el que se contrata al  $i$ -ésimo candidato. De este modo,

$X_i = 1$  si el candidato  $i$  es contratado

$0$  si el candidato  $i$  no es contratado

y

$$\sum_{i=1}^n X_i = \sum_{i=1}^n \Pr \{X_i=1\} \quad (5.2)$$

Por el Lema 5.1 tenemos que

$E \text{CEx}_i D \Pr f_{\text{candidato } i} \text{ es contratado};$

y, por lo tanto, debemos calcular la probabilidad de que se ejecuten las líneas 5 y 6 de HIRE-ASSISTANT .

El candidato  $i$  es contratado, en la línea 6, exactamente cuando el candidato  $i$  es mejor que cada uno de los candidatos 1 a  $i - 1$ . Como hemos supuesto que los candidatos llegan en orden aleatorio, los primeros  $i$  candidatos han aparecido en un orden aleatorio. Cualquiera de estos primeros  $i$  candidatos tiene la misma probabilidad de ser el mejor calificado hasta el momento. El candidato  $i$  tiene una probabilidad de  $1 = i$  de estar mejor calificado que los candidatos 1 a  $i - 1$  y, por lo tanto, una probabilidad de  $1 = i$  de ser contratado. Por el Lema 5.1, concluimos que

$$E \text{CEx}_i D 1 = i \quad (5.3)$$

Ahora podemos calcular  $E \text{CEx}$ :

$$E \text{CEx} DE " Xn Xi #; \text{por ecuación (5.2)} \quad (5.4)$$

$$\begin{aligned} Xn &= E \text{CEx}_i \\ &\text{por linealidad de la expectativa} \\ &iD1 \end{aligned}$$

$$\begin{aligned} Xn &= 1 = yo \\ &\text{por la ecuación (5.3)} \\ &iD1 \\ &D \ln n C O.1 / \text{por ecuación (A.7)}. \quad (5.5) \end{aligned}$$

Aunque entrevistamos a  $n$  personas, en realidad contratamos solo aproximadamente a  $\ln n$  de ellas, en promedio. Resumimos este resultado en el siguiente lema.

### Lema 5.2

Suponiendo que los candidatos se presentan en un orden aleatorio, el algoritmo CONTRATAR ASISTENTE tiene un costo de contratación total promedio por caso de  $O.ch \ln n/$ .

Prueba El límite se sigue inmediatamente de nuestra definición del costo de contratación y la ecuación (5.5), que muestra que el número esperado de contrataciones es aproximadamente  $\ln n$ . ■

El costo de contratación del caso promedio es una mejora significativa sobre el costo de contratación del peor de los casos de  $O.chn/$ .

### Ejercicios

5.2-1

En HIRE-ASSISTANT, suponiendo que los candidatos se presentan en orden aleatorio, ¿cuál es la probabilidad de que contrate exactamente una vez? ¿Cuál es la probabilidad de que contrate exactamente n veces?

5.2-2

En HIRE-ASSISTANT, suponiendo que los candidatos se presentan en orden aleatorio, ¿cuál es la probabilidad de que contrate exactamente dos veces?

5.2-3

Utilice variables aleatorias indicadoras para calcular el valor esperado de la suma de n dados.

5.2-4

Use variables aleatorias indicadoras para resolver el siguiente problema, que se conoce como el problema de la verificación del sombrero. Cada uno de los n clientes le da un sombrero a una persona que guarda sombreros en un restaurante. La persona que revisa los sombreros devuelve los sombreros a los clientes en un orden aleatorio. ¿Cuál es el número esperado de clientes que recuperan su propio sombrero?

5.2-5

Sea  $A \in \mathbb{R}^{1 : n}$  un arreglo de n números distintos. Si  $i < j$  y  $A[i] > A[j]$ , entonces el par  $(i, j)$  se denomina inversión de A. (Consulte el problema 2-4 para obtener más información sobre las inversiones). Suponga que los elementos de A forman una permutación aleatoria uniforme de  $h[1; 2; \dots; n]$ . Utilice variables aleatorias indicadoras para calcular el número esperado de inversiones

---

## 5.3 Algoritmos aleatorios

En la sección anterior, mostramos cómo conocer una distribución en las entradas puede ayudarnos a analizar el comportamiento de caso promedio de un algoritmo. Muchas veces, no tenemos ese conocimiento, lo que impide un análisis de caso promedio. Como se mencionó en la Sección 5.1, es posible que podamos usar un algoritmo aleatorio.

Para un problema como el de contratación, en el que es útil suponer que todas las permutaciones de la entrada son igualmente probables, un análisis probabilístico puede guiar el desarrollo de un algoritmo aleatorio. En lugar de suponer una distribución de insumos, imponemos una distribución. En particular, antes de ejecutar el algoritmo, permutamos aleatoriamente a los candidatos para hacer cumplir la propiedad de que todas las permutaciones son igualmente probables. Aunque hemos modificado el algoritmo, todavía esperamos contratar a un nuevo asistente de oficina aproximadamente ln n veces. Pero ahora esperamos

este sería el caso para cualquier entrada, en lugar de para las entradas extraídas de una distribución particular.

Exploraremos más a fondo la distinción entre el análisis probabilístico y los algoritmos aleatorios. En la Sección 5.2, afirmamos que, suponiendo que los candidatos llegan en un orden aleatorio, el número esperado de veces que contratamos a un nuevo asistente de oficina es aproximadamente  $\ln n$ . Tenga en cuenta que el algoritmo aquí es determinista; para cualquier entrada en particular, la cantidad de veces que se contrata a un nuevo asistente de oficina es siempre la misma. Además, la cantidad de veces que contratamos a un nuevo asistente de oficina difiere según las distintas entradas y depende de los rangos de los distintos candidatos. Dado que este número depende únicamente de los rangos de los candidatos, podemos representar una entrada particular enumerando, en orden, los rangos de los candidatos, es decir,  $h_{rank.1}/; rango.2/; \dots; rango.n/i$ . Dada la lista de rango A1 D h1; 2; 3; 4; 5; 6; 7; 8; 9; 10i, siempre se contrata un nuevo asistente de oficina 10 veces, ya que cada candidato sucesivo es mejor que el anterior, y las líneas 5 y 6 se ejecutan en cada iteración. Dada la lista de rangos A2 D h10; 9; 8; 7; 6; 5; 4; 3; 2; 1i, un nuevo asistente de oficina se contrata solo una vez, en la primera iteración. Dada una lista de rangos A3 D h5; 2; 1; 8; 4; 7; 10; 9; 3; 6i, se contrata tres veces a un nuevo asistente de oficina, al entrevistar a los candidatos con los rangos 5, 8 y 10. Recordando que el costo de nuestro algoritmo depende de cuántas veces contratamos a un nuevo asistente de oficina, vemos que hay insumos costosos como A1, insumos económicos como A2 e insumos moderadamente costosos como A3.

Considere, por otro lado, el algoritmo aleatorio que primero permuta los candidatos y luego determina el mejor candidato. En este caso, aleatorizamos en el algoritmo, no en la distribución de entrada. Dada una entrada en particular, digamos A3 arriba, no podemos decir cuántas veces se actualiza el máximo, porque esta cantidad difiere con cada ejecución del algoritmo. La primera vez que ejecutamos el algoritmo en A3, puede producir la permutación A1 y realizar 10 actualizaciones; pero la segunda vez que ejecutamos el algoritmo, podemos producir la permutación A2 y realizar solo una actualización. La tercera vez que lo ejecutamos, podemos realizar alguna otra cantidad de actualizaciones.

Cada vez que ejecutamos el algoritmo, la ejecución depende de las elecciones aleatorias realizadas y es probable que difiera de la ejecución anterior del algoritmo. Para este algoritmo y muchos otros algoritmos aleatorios, ninguna entrada en particular provoca el comportamiento del peor de los casos. Incluso su peor enemigo no puede producir una matriz de entrada incorrecta, ya que la permutación aleatoria hace que el orden de entrada sea irrelevante. El algoritmo aleatorio funciona mal solo si el generador de números aleatorios produce una permutación "desafortunada".

Para el problema de contratación, el único cambio necesario en el código es silenciar aleatoriamente la matriz.

### ASISTENTE DE CONTRATACIÓN

ALEATORIA.n/ 1 permutar aleatoriamente la lista  
 de candidatos 2 mejor D 0 // el candidato 0 es un candidato ficticio menos calificado 3  
 para i D 1 a n 4  
 entrevistar al candidato i 5 si el  
 candidato i es mejor que el mejor candidato 6 mejores D i 7  
 candidatos a contratar i

Con este simple cambio, hemos creado un algoritmo aleatorio cuyo rendimiento coincide con el obtenido al suponer que los candidatos se presentaron en un orden aleatorio.

Lema 5.3 El

costo esperado de contratación del procedimiento ASISTENTE DE CONTRATACIÓN ALEATORIA es  $O(\ln n)$ .

Prueba Después de permutar la matriz de entrada, hemos logrado una situación idéntica a la del análisis probabilístico de HIRE-ASSISTANT. ■

La comparación de los lemas 5.2 y 5.3 destaca la diferencia entre el análisis probabilístico y los algoritmos aleatorios. En el Lema 5.2, hacemos una suposición sobre la entrada. En el Lema 5.3, no hacemos tal suposición, aunque la aleatorización de la entrada lleva un tiempo adicional. Para permanecer consistentes con nuestra terminología, expresamos el Lema 5.2 en términos del costo de contratación de un caso promedio y el Lema 5.3 en términos del costo de contratación esperado. En el resto de esta sección, analizamos algunos problemas relacionados con la permutación aleatoria de entradas.

### Arreglos de permutación aleatoria

Muchos algoritmos aleatorios aleatorizan la entrada permutando la matriz de entrada dada. (Hay otras formas de usar la aleatorización.) Aquí, discutiremos dos métodos para hacerlo. Suponemos que tenemos un arreglo A que, sin pérdida de generalidad, contiene los elementos 1 a n. Nuestro objetivo es producir una permutación aleatoria de la matriz.

Un método común es asignar a cada elemento  $A[i]$  de la matriz una prioridad aleatoria  $P[i]$  y luego clasificar los elementos de A de acuerdo con estas prioridades. Por ejemplo, si nuestro arreglo inicial es  $A = [1, 2, 3, 4]$  y elegimos prioridades aleatorias  $P = [0.2, 0.5, 0.1, 0.4]$ , produciríamos un arreglo  $B = [3, 1, 4, 2]$ , ya que la segunda prioridad es la más pequeña, seguida de la cuarta, luego la primera y finalmente la tercera.

Llamamos a este procedimiento PERMUTE-BY-SORTING:

PERMUTE-BY-SORTING.A/ 1 n D

A:longitud 2 sea P

Œ1 : : n un nuevo arreglo 3 para i D

1 an P Œi D

RANDOM.1; n3/ 4 5 ordenar

A, usando P como claves de ordenación

La línea 4 elige un número aleatorio entre 1 y n3. Usamos un rango de 1 a n3 para que sea probable que todas las prioridades en P sean únicas. (El ejercicio 5.3-5 le pide que demuestre que la probabilidad de que todas las entradas sean únicas es al menos  $1 - \frac{1}{n!}$ , y el ejercicio 5.3-6 pregunta cómo implementar el algoritmo incluso si dos o más prioridades son idénticas). Supongamos que todas las prioridades son únicas.

El paso que lleva mucho tiempo en este procedimiento es la clasificación en la línea 5. Como veremos en el Capítulo 8, si usamos una clasificación por comparación, la clasificación lleva  $.n \lg n$  de tiempo. Podemos lograr este límite inferior, ya que hemos visto que la ordenación por fusión toma  $.n \lg n$  de tiempo. (Veremos otros géneros de comparación que toman  $.n \lg n$  tiempo en la Parte II). El ejercicio 8.3-4 le pide que resuelva el problema muy similar de ordenar números en el rango de 0 a n3 1 en el tiempo  $O(n^2)$ . Después de ordenar, si  $P[i]$  es la  $j$ -ésima prioridad más pequeña, entonces  $A[i]$  se encuentra en la posición  $j$  de la salida  $A$ . De esta manera obtenemos una permutación. Queda por demostrar que el procedimiento produce una permutación aleatoria uniforme, es decir, que es igualmente probable que el procedimiento produzca todas las permutaciones de los números 1 a n.

Lema 5.4 El

procedimiento PERMUTE-BY-SORTING produce una permutación aleatoria uniforme de la entrada, asumiendo que todas las prioridades son distintas.

Prueba Comenzamos considerando la permutación particular en la que cada elemento  $A[i]$  recibe la  $i$ -ésima prioridad más pequeña. Mostraremos que esta permutación ocurre con probabilidad exactamente  $1/n!$ . Para  $i \in \{1, 2, \dots, n\}$ , sea  $E_i$  el evento de que el elemento  $A[i]$  reciba la  $i$ -ésima prioridad más pequeña. Entonces deseamos calcular la probabilidad de que para todo  $i$  ocurra el evento  $E_i$ , que es

$\Pr[f(E_1 \wedge E_2 \wedge E_3) \wedge \dots \wedge E_n] = \prod_{i=1}^n \Pr[E_i]$

Usando el Ejercicio C.2-5, esta probabilidad es igual a

$\Pr[f(E_1)] \Pr[f(E_2)] \dots \Pr[f(E_n)] = \frac{1}{n!} \Pr[f(E_1)]$

$\Pr[f(E_1)] = \Pr[f(E_1 \wedge E_2 \wedge E_3 \wedge \dots \wedge E_n)] = \Pr[f(E_1)] \Pr[f(E_2 \wedge E_3 \wedge \dots \wedge E_n)]$

Tenemos que  $\Pr[f(E_1)] = 1/n$  porque es la probabilidad de que una prioridad elegida al azar de un conjunto de n sea la prioridad más pequeña. A continuación, observamos

que  $\Pr fE2 \mid E1g D 1 = .n 1 /$  porque dado que el elemento  $A\in 1$  tiene la prioridad más pequeña, cada uno de los  $n - 1$  elementos restantes tiene la misma probabilidad de tener la segunda prioridad más pequeña. En general, para  $i \in D; 2; 3; \dots; n$ , tenemos que  $\Pr fEi \mid E1 \setminus Ei2 \setminus E1g D 1 = .n_i C 1 /$ , ya que, dado que los elementos  $A\in 1$  a  $A\in i$  tienen las  $i - 1$  prioridades más pequeñas (en orden), cada uno de los restantes  $n - i + 1$  elementos tiene la misma posibilidad de tener la  $i$ -ésima prioridad más pequeña. De este modo, tenemos

$$\Pr fE1 \setminus E2 \setminus E3 \setminus E1g D = \frac{1}{n} \cdot \frac{1}{n-1} \cdot \frac{1}{n-2} \cdot \frac{1}{n-3} \cdots$$

y hemos demostrado que la probabilidad de obtener la permutación identidad es  $1 = n! /$

Podemos extender esta prueba para trabajar con cualquier permutación de prioridades. Considere cualquier permutación fija  $D = h_1, h_2, \dots, h_n$  del conjunto  $f = 1, 2, \dots, n$ . Denotemos por  $r_i$  el rango de prioridad asignado al elemento  $A\in i$ , donde el elemento con la  $j$ -ésima prioridad tiene rango  $j$ . Si definimos  $E_i$  como el evento en el que el elemento  $A\in i$  recibe la  $i$ -ésima prioridad más pequeña, o  $r_i = h_i$ , se aplica la misma demostración. Por lo tanto, si calculamos la probabilidad de obtener alguna permutación en particular, el cálculo es idéntico al anterior, por lo que la probabilidad de obtener esta permutación también es  $1 = n! /$  ■

Se podría pensar que para probar que una permutación es una permutación aleatoria uniforme, basta con mostrar que, para cada elemento  $A\in i$ , la probabilidad de que el elemento termine en la posición  $j$  es  $1/n$ . El ejercicio 5.3-4 muestra que esta condición más débil es, de hecho, insuficiente.

Un mejor método para generar una permutación aleatoria es permutar la matriz dada en su lugar. El procedimiento RANDOMIZE-IN-PACE lo hace en  $O(n^2)$  tiempo. En su  $i$ -ésima iteración, elige el elemento  $A\in i$  aleatoriamente entre los elementos  $A\in 1$  a  $A\in n$ . Posterior a la  $i$ -ésima iteración,  $A\in i$  nunca se altera.

#### RANDOMIZE-IN-PACE.A/ 1 n

```
D A:longitud 2 para
i D 1 a n
3      intercambiar A\in i con A\in RANDOM.i; norte/
```

Usaremos un bucle invariante para mostrar que el procedimiento RANDOMIZE-IN-PACE produce una permutación aleatoria uniforme. Una  $k$ -permutación en un conjunto de  $n$  elementos es una secuencia que contiene  $k$  de los  $n$  elementos, sin repeticiones. (Consulte el Apéndice C.) Hay  $n! / (n-k)!$  tales posibles  $k$ -permutaciones.

Lema 5.5 El

procedimiento RANDOMIZE-IN-PACE calcula una permutación aleatoria uniforme.

Prueba Usamos el siguiente bucle invariante:

Justo antes de la  $i$ -ésima iteración del ciclo for de las líneas 2–3, para cada posible  $.i$  1-/permutación de los  $n$  elementos, el subarreglo  $A[1 : : i]$  contiene esta  $i$  1-/permutación con probabilidad  $.ni C 1 /S = nS$ .

Necesitamos mostrar que esta invariante es verdadera antes de la primera iteración del ciclo, que cada iteración del ciclo mantiene la invariante y que la invariante proporciona una propiedad útil para mostrar la corrección cuando termina el ciclo.

Inicialización: considere la situación justo antes de la primera iteración del bucle, de modo que  $i = 1$ . El invariante del bucle dice que para cada posible permutación 0, el subconjunto  $A[1 : : 0]$  contiene esta permutación 0 con probabilidad  $.ni C 1 /S = nS = 1$ . El subarreglo  $A[1 : : 0]$  es un subarreglo vacío, y una permutación 0 no tiene elementos. Por lo tanto,  $A[1 : : 0]$  contiene cualquier permutación de 0 con probabilidad 1, y el ciclo invariante se cumple antes de la primera iteración.

Mantenimiento: Suponemos que justo antes de la  $i$ -ésima iteración, cada posible  $i$  1-/permutación aparece en el subarreglo  $A[1 : : i]$  con probabilidad  $.ni C 1 /S = nS$ , y mostraremos que después de la  $i$ -ésima iteración, cada posible  $i$ -permutación aparece en el subarreglo  $A[1 : : i]$  con probabilidad  $.ni C 1 /S = nS$ .

Incrementar  $i$  para la siguiente iteración mantiene el bucle invariable.

Examinemos la  $i$ -ésima iteración. Considere una permutación  $i$  particular y denote los elementos en ella por  $x_1; x_2; \dots; x_i$ . Esta permutación consiste en una permutación  $.i$  1-/  $x_1; \dots; x_i$  seguida del valor  $x_i$  que el algoritmo coloca en  $A[i]$ . Sea  $E_1$  el evento en el cual el primer  $i$  creó la particular  $i$  1-/permutación  $x_1; \dots; x_i$  en  $A[1 : : i]$ . 1 iteraciones tienen 1. Por el bucle invariante,  $\Pr f(E_1) = .ni C 1 /S = nS$ . Sea  $E_2$  el evento de que  $i$ -ésima iteración pone a  $x_i$  en la posición  $A[i]$ . La  $i$ -permutación  $x_1; \dots; x_i$  aparece en  $A[1 : : i]$  precisamente cuando ocurren tanto  $E_1$  como  $E_2$ , por lo que deseamos calcular  $\Pr f(E_2 | E_1)$ .

Usando la ecuación (C.14), tenemos

$\Pr f(E_2 | E_1) = \Pr f(E_2) / \Pr f(E_1)$

La probabilidad  $\Pr f(E_2 | E_1)$  es igual a  $1 = .ni C 1 /S = nS$  porque en la línea 3 el algoritmo elige aleatoriamente  $x_i$  de los valores de  $ni C 1$  en las posiciones  $A[1 : : n]$ . Así, tenemos

$$\Pr fE2 \setminus E1g D \Pr fE2 j E1g \Pr fE1g .ni C 1/\check{S}$$

$$D \frac{1}{ni C 1 .ni /S} = \frac{1}{n\check{S}}$$

$$D \frac{1}{n\check{S}}$$

Terminación: En la terminación,  $i \in D$   $n \in C 1$ , y tenemos que el subarreglo  $A[1:n]$  es una  $n$ -permutación dada con probabilidad  $.n.nC1/C1=n\check{S}$   $D 0\check{S}=n\check{S}$   $D 1=n\check{S}$ .

notas

Por lo tanto, RANDOMIZE-IN-PACE produce una permutación aleatoria uniforme. ■

Un algoritmo aleatorio es a menudo la forma más simple y eficiente de resolver un problema. Usaremos algoritmos aleatorios ocasionalmente a lo largo de este libro.

### Ejercicios

#### 5.3-1

El profesor Marceau se opone al invariante de bucle utilizado en la prueba del Lema 5.5. Se pregunta si es cierto antes de la primera iteración. Él razona que podríamos declarar fácilmente que un subarreglo vacío no contiene permutaciones 0. Por lo tanto, la probabilidad de que un subarreglo vacío contenga una permutación 0 debería ser 0, lo que invalidaría el bucle invariante antes de la primera iteración. Reescriba el procedimiento RANDOMIZE-IN-PACE de modo que su bucle invariante asociado se aplique a un subarreglo no vacío antes de la primera iteración y modifique la prueba del Lema 5.5 para su procedimiento.

#### 5.3-2

El profesor Kelp decide escribir un procedimiento que produzca al azar cualquier permutación además de la permutación de identidad. Propone el siguiente procedimiento:

**PERMUTA-SIN-IDENTIDAD.A/ 1 n D A:longitud**

2 para  $i \in D$  1 a  $n - 1$  3

intercambiar  $A[i]$  con  $A[RANDOM(i, n)]$

¿Este código hace lo que pretende el profesor Kelp?

#### 5.3-3

Suponga que en lugar de intercambiar el elemento  $A[i]$  con un elemento aleatorio del subarreglo  $A[i:n]$ , lo intercambiamos con un elemento aleatorio de cualquier parte del arreglo:

PERMUTAR-CON-TODO.A/ 1

```
n D A:longitud 2 para
i D 1 a n 3 intercambiar
    AŒi con AŒRANDOM.1; norte/
```

¿Este código produce una permutación aleatoria uniforme? ¿Por qué o por qué no?

5.3-4

El profesor Armstrong sugiere el siguiente procedimiento para generar una permutación aleatoria uniforme:

PERMUTE-BY-CYCLIC.A/ 1 n D

```
A:longitud 2 sea
BŒ1 : : n un nuevo arreglo 3 offset
D RANDOM.1; n/ 4 para i D 1 a n
5 6
    dest D i C offset si
    dest > n dest
7         D dest n
8         BŒdest D AŒi 9
retorno B
```

Demuestre que cada elemento AŒi tiene una probabilidad de 1=n de terminar en cualquier posición particular en B. Luego demuestre que el profesor Armstrong se equivoca al mostrar que la permutación resultante no es uniformemente aleatoria.

5.3-5 ?

Demuestre que en el arreglo P en el procedimiento PERMUTE-BY-SORTING, la probabilidad de que todos los elementos sean únicos es al menos 1 1=n.

5.3-6

Explique cómo implementar el algoritmo PERMUTE-BY-SORTING para manejar el caso en el que dos o más prioridades son idénticas. Es decir, su algoritmo debe producir una permutación aleatoria uniforme, incluso si dos o más prioridades son idénticas.

5.3-7

Supongamos que queremos crear una muestra aleatoria del conjunto f1; 2; 3; : : : ; ng, es decir, un subconjunto S de m elementos, donde 0 mn, de modo que es igualmente probable que se cree cada m subconjunto. Una forma sería establecer AŒi D i para i D 1; 2; 3; : : : ; n, llame a RANDOMIZE-IN-PLACE.A/, y luego tome solo los primeros m elementos de la matriz.

Este método haría n llamadas al procedimiento RANDOM . Si n es mucho mayor que m, podemos crear una muestra aleatoria con menos llamadas a RANDOM. Muestra esa

el siguiente procedimiento recursivo devuelve un m-subconjunto aleatorio S de f1; 2; 3; : : : ; ng, en el que cada subconjunto m es igualmente probable, mientras que solo se realizan m llamadas a RANDOM:

```
MUESTRA ALEATORIA.m; n/ 1
si m == 0 2
volver ; 3 más SD
RANDOM-SAMPLE.m 1; n 1/ 4 i D ALEATORIO.1; n/ si yo 2
S
5
6           SDS [ fng else
7           SDS [ fig return S
8
```

## ? 5.4 Análisis probabilístico y otros usos de variables aleatorias indicadoras

Esta sección avanzada ilustra aún más el análisis probabilístico por medio de cuatro ejemplos. El primero determina la probabilidad de que en una habitación de k personas, dos de ellas comparten el mismo cumpleaños. El segundo ejemplo examina lo que sucede cuando arrojamos bolas al azar en contenedores. El tercero investiga “rachas” de caras consecutivas cuando lanzamos monedas. El ejemplo final analiza una variante del problema de contratación en el que debe tomar decisiones sin entrevistar realmente a todos los candidatos.

### 5.4.1 La paradoja del cumpleaños

Nuestro primer ejemplo es la paradoja del cumpleaños. ¿Cuántas personas debe haber en una habitación para que haya un 50% de probabilidad de que dos de ellos hayan nacido el mismo día del año? La respuesta es sorprendentemente pocas. La paradoja es que, de hecho, es mucho menor que el número de días de un año, o incluso la mitad del número de días de un año, como veremos.

Para responder a esta pregunta, indexamos a las personas en la habitación con los números enteros 1; 2; : : : ; k, donde k es el número de personas en la habitación. Ignoramos el tema de los años bisiestos y asumimos que todos los años tienen n D 365 días. Para i D 1; 2; : : : ; k, sea bi el día del año en que cae el cumpleaños de la persona i, donde 1 También suponemos que los cumpleaños se distribuyen uniformemente entre los n días del año, de modo que Pr fbi D rg D 1=n para i D 1; 2; : : : ; kyr D 1; 2; : : : ; La probabilidad de que dos personas dadas, digamos i y j, tengan cumpleaños coincidentes depende de si la selección aleatoria de cumpleaños es independiente. Suponemos de ahora en adelante que los cumpleaños son independientes, por lo que la probabilidad de que yo sea el cumpleaños

y el cumpleaños de j ambos caen en el día r es

$$\Pr \{fbi \in D \text{ y } bj \in D\} = \Pr \{fbi \in D\} \cdot \Pr \{bj \in D\}$$

$$D1=n2 :$$

Por lo tanto, la probabilidad de que ambos caigan el mismo día es

$$\Pr \{fbi \in D \text{ y } bj \in D\} = \frac{\Pr \{fbi \in D\} \cdot \Pr \{bj \in D\}}{rD1}$$

$$Xn = \frac{.1=n2}{rD1}$$

$$D1=n : \quad (5.6)$$

Más intuitivamente, una vez que se elige  $b_i$ , la probabilidad de que se elija  $b_j$  para el mismo día es  $1/n$ . Por lo tanto, la probabilidad de que i y j tengan el mismo cumpleaños es la misma que la probabilidad de que el cumpleaños de uno de ellos caiga en un día determinado.

Nótese, sin embargo, que esta coincidencia depende de la suposición de que los cumpleaños son independientes.

Podemos analizar la probabilidad de que al menos 2 de k personas tengan cumpleaños coincidentes observando el evento complementario. La probabilidad de que al menos dos de los cumpleaños coincidan es 1 menos la probabilidad de que todos los cumpleaños sean diferentes. El evento de que k personas tengan distintos cumpleaños es

$$Bk \in D \setminus \bigcup_{i=1}^k A_i$$

donde  $A_i$  es el evento de que el cumpleaños de la persona  $i$  es diferente al de la persona  $j$  para todos los  $j < i$ . Como podemos escribir  $Bk \in D \setminus \bigcup_{i=1}^k A_i$ , obtenemos de la ecuación (C.16) la recurrencia

$$\Pr \{fBkg \in D\} = \Pr \{fBk1g \in D\} \cdot \Pr \{fAk \in D \mid Bk1g \in D\} ; \quad (5.7)$$

donde tomamos  $\Pr \{fB1g \in D\} = \Pr \{fA1g \in D\} = 1$  como condición inicial. En otras palabras, la probabilidad de que  $b_1, b_2, \dots, b_k$  son cumpleaños distintos es la probabilidad de que  $b_1, b_2, \dots, b_k$  son cumpleaños distintos multiplicados por la probabilidad de que  $b_k \neq b_i$  para  $k \geq 1$ , dado que  $b_1, b_2, \dots, b_{k-1}$

son distintos. Si  $b_1, b_2, \dots, b_k$  son distintos, la probabilidad condicional de que  $b_k \neq b_i$  para  $k \geq 1$  es  $\Pr \{fAk \in D \mid Bk1g \in D\} = nk/(n-k)$ , ya que de los  $n$  días,  $n-k$  días no se toman. Aplicamos iterativamente la recurrencia (5.7) para obtener

$$\Pr(fBkg) D \Pr(fBk1g) \Pr(fak) j Bk1g \\ D \Pr(fBk2g) \Pr(fAk1) j Bk2g \Pr(fAk) j Bk1g$$

⋮

$$D \Pr(fB1g) \Pr(fA2) j B1g \Pr(fA3) j B2g \Pr(fak) j Bk1g nk C 1$$

	$n 1$	$n^o 2$		
$D 1$	$\overline{1}$	$\overline{1}$	$\overline{1}$	$\overline{1}$
	norte	norte	norte	norte
$D 1$	1	1	2	1
	norte	norte	norte	norte
	$\overline{1}$	$\overline{2}$	$\overline{1}$	$\overline{k 1}$

La desigualdad (3.12),  $1 C \times ex$ , nos da

$$\Pr(fBkg) \quad e1=ne2=n \quad e.k1/n = D \Pr(fPk1)$$

$$iD1 i=n D ek.k1/$$

$$=2n$$

$$1=2$$

cuando  $kk 1/2n ln.1=2/$ . La probabilidad de que todos los  $k$  cumpleaños sean distintos es como máximo  $1=2$  cuando  $kk 1/2n ln 2$  o, resolviendo la ecuación cuadrática, cuando  $k .1 C p1 C .8 ln 2/n=2$ . Para  $n D 365$ , debemos tener  $k 23$ . Así, si al menos 23 personas están en una habitación, la probabilidad es al menos  $1=2$  de que al menos dos personas tengan el mismo cumpleaños. En Marte, un año tiene 669 días marcianos; por lo tanto, se necesitan 31 marcianos para obtener el mismo efecto.

Un análisis usando variables aleatorias indicadoras

Podemos usar variables aleatorias indicadoras para proporcionar un análisis más simple pero aproximado de la paradoja del cumpleaños. Para cada par  $i, j$  de las  $k$  personas en la sala, definimos la variable aleatoria indicadora  $X_{ij}$ , para  $1 < j, k$ , por

$X_{ij} D 1$  persona  $i$  y persona  $j$  tienen el mismo cumpleaños

$D (1$  si la persona  $i$  y la persona  $j$  tienen el mismo cumpleaños;

Por la ecuación (5.6), la probabilidad de que dos personas tengan el mismo cumpleaños es  $1=n$  y, por lo tanto, por el Lema 5.1, tenemos

$E \sum X_{ij} D \Pr(f persona i y persona j tienen el mismo cumpleaños)$

$D1=n:$

Si  $X$  es la variable aleatoria que cuenta el número de pares de individuos que tienen el mismo cumpleaños, tenemos

$$\sum_{i=1}^k \sum_{j=1}^k X_{ij} = \sum_{i=1}^k \sum_{j=1}^k \mathbb{E}[X_{ij}]$$

Tomando las expectativas de ambos lados y aplicando la linealidad de la expectativa, obtenemos

$$\begin{aligned} & \sum_{i=1}^k \sum_{j=1}^k \mathbb{E}[X_{ij}] = \sum_{i=1}^k \sum_{j=1}^k \mathbb{E}[X_{ij}] \\ & \sum_{i=1}^k \sum_{j=1}^k \mathbb{E}[X_{ij}] = \sum_{i=1}^k \sum_{j=1}^k \frac{1}{n} \\ & \sum_{i=1}^k \sum_{j=1}^k \frac{1}{n} = \frac{k(k-1)}{2n} \end{aligned}$$

Cuando  $k(k-1)/2n \geq 1$ , por lo tanto, el número esperado de parejas de personas con el mismo cumpleaños es al menos 1. Así, si tenemos al menos  $p2nC1$  individuos en una habitación, podemos esperar que al menos dos tengan el mismo cumpleaños. Para  $n=365$ , si  $k=28$ , el número esperado de parejas con el mismo cumpleaños es  $.28 \cdot 27 / (2 \cdot 365) \approx 0.0356$ . Por lo tanto, con al menos 28 personas, esperamos encontrar al menos un par de días de cumpleaños coincidentes. En Marte, donde un año tiene 669 días marcianos, necesitamos al menos 38 marcianos.

El primer análisis, que usó solo probabilidades, determinó el número de personas requeridas para que la probabilidad de que exista un par de cumpleaños coincidentes exceda  $1-2^{-k}$ , y el segundo análisis, que usó variables aleatorias indicadoras, determinó el número tal que el esperado número de cumpleaños coincidentes es 1. Aunque los números exactos de personas difieren para las dos situaciones, son los mismos asintóticamente:  $\sqrt{pn}$ .

#### 5.4.2 Bolas y contenedores

Considere un proceso en el que lanzamos al azar bolas idénticas en b contenedores, numerados 1, 2, ..., b. Los lanzamientos son independientes, y en cada lanzamiento la bola tiene la misma probabilidad de 1/b de caer en cualquier recipiente. La probabilidad de que una bola lanzada caiga en cualquier recipiente dado es 1/b. Por lo tanto, el proceso de lanzamiento de la pelota es una secuencia de intentos de Bernoulli (consulte el Apéndice C.4) con una probabilidad de éxito de 1/b, donde el éxito significa que la pelota cae en el recipiente dado. Este modelo es particularmente útil para analizar el hashing (vea el Capítulo 11), y podemos responder una variedad de preguntas interesantes sobre el proceso de lanzar una pelota. (El problema C-1 hace preguntas adicionales sobre pelotas y recipientes).

¿Cuántas bolas caen en un contenedor dado? El número de bolas que caen en un contenedor dado sigue la distribución binomial  $b \sim \text{Bin}(n; 1=b)$ . Si lanzamos  $n$  bolas, la ecuación (C.37) nos dice que el número esperado de bolas que caen en el contenedor dado es  $n=b$ .

¿Cuántas pelotas debemos lanzar, en promedio, hasta que un recipiente dado contenga una pelota? El número de lanzamientos hasta que el recipiente dado recibe una pelota sigue la distribución geométrica con probabilidad  $1=b$  y, por la ecuación (C.32), el número esperado de lanzamientos hasta el éxito es  $1/(1-b)$ .

¿Cuántas bolas debemos lanzar hasta que cada contenedor contenga al menos una bola? Llamemos "golpe" a un lanzamiento en el que una bola cae en un recipiente vacío. Queremos saber el número esperado  $n$  de lanzamientos necesarios para obtener  $b$  aciertos.

Usando los aciertos, podemos dividir los  $n$  lanzamientos en etapas. La  $i$ -ésima etapa consiste en los lanzamientos después del  $i$ -ésimo golpe hasta el  $i$ -ésimo golpe. La primera etapa consiste en el primer lanzamiento, ya que tenemos la garantía de acertar cuando todas las papeleras estén vacías. Para cada lanzamiento durante la  $i$ -ésima etapa, los contenedores  $i$  contienen bolas y los contenedores  $i+1$  están vacíos. Así, para cada lanzamiento en la  $i$ -ésima etapa, la probabilidad de obtener un acierto es  $b/(i+1)$ .

Sea  $n_i$  el número de lanzamientos en la  $i$ -ésima etapa. Por tanto, el número de lanzamientos necesarios para obtener  $b$  aciertos es  $n = n_1 + n_2 + \dots + n_b$ . Cada variable aleatoria  $n_i$  tiene una distribución geométrica con probabilidad de éxito  $b/(i+1)$  y así, por la ecuación (C.32), tenemos

$$E(n_i) = \frac{b}{b/(i+1)} = i+1$$

Por linealidad de la expectativa, tenemos

$b$

$$E(n) = E(n_1 + n_2 + \dots + n_b) =$$

$b$

$$E(n_1) = \frac{b}{b/(1+1)} = 2$$

$b$

$$E(n_2) = \frac{b}{b/(2+1)} = 3$$

$b$

$$E(n_b) = \frac{b}{b/(b+1)} = b+1$$

$$\text{re } b \times \frac{1}{i}$$

$$\text{re } b \times \frac{1}{i}$$

$$= b \ln b + b \ln(b+1) - b \ln b = b \ln(b+1)$$

Por lo tanto, toma aproximadamente  $b \ln b$  lanzamientos antes de que podamos esperar que cada contenedor tenga una pelota. Este problema también se conoce como el problema del coleccionista de cupones, que dice que una persona que intenta recolectar cada uno de  $b$  cupones diferentes espera adquirir aproximadamente  $b \ln b$  cupones obtenidos al azar para tener éxito.

### 5.4.3 Rayas

Supón que lanzas una moneda justa  $n$  veces. ¿Cuál es la racha más larga de cabezas consecutivas que esperas ver? La respuesta es  $\lg n$ , como el siguiente análisis

Primero probamos que la longitud esperada de la racha más larga de caras es  $O(\lg n)$ . La probabilidad de que en cada lanzamiento de moneda salga cara es  $1/2$ . Sea  $A_{ik}$  el evento de que una racha de caras de longitud al menos  $k$  comience con el  $i$ -ésimo lanzamiento de la moneda o, más precisamente, el evento de que la  $k$  moneda consecutiva se lance  $i$ ; y  $C_1; \dots; C_k$  dan solo caras, donde  $C_1$  y  $C_k$  son los resultados de los lanzamientos de monedas. Dado que los lanzamientos de monedas son mutuamente independientes, para cualquier evento dado  $A_{ik}$ , la probabilidad de que todos los  $k$  lanzamientos sean caras es

$$\Pr[\text{faikg } D \mid I=2k] \quad (5.8)$$

Para k D 2 dlg ne,

Pr fAi;2dlq neq D 1=22dlqne 1=22

$\lg n$

D1=n2;

y por lo tanto, la probabilidad de que una racha de cabezas de al menos 2 dígitos comience en la posición  $i$  es bastante pequeña. Hay como máximo  $n - 2$  posiciones donde tal racha puede comenzar. Por lo tanto, la probabilidad de que una racha de cabezas de al menos 2 dígitos comience en cualquier lugar es

$$\begin{aligned}
 & \text{lgneC1} & n2 & \text{dlg neC1} \\
 & \Pr(n2diD1 \quad Ai;2\text{dlg ne }) & X & 1=n2 \\
 & & iD1 & \\
 & & & \\
 & & & \\
 & & & \\
 & & & \\
 & < Xn & 1=n2 \\
 & iD1 & \\
 & D1=n; & & (5.9)
 \end{aligned}$$

ya que por la desigualdad de Boole (C.19), la probabilidad de una unión de eventos es como máximo la suma de las probabilidades de los eventos individuales. (Tenga en cuenta que la desigualdad de Boole se cumple incluso para eventos como estos que no son independientes).

Ahora usamos la desigualdad (5.9) para acotar la longitud de la racha más larga. Para  $j \in D(0; 1; 2; \dots; n)$ , sea  $L_j$  el evento de que la racha más larga de cabezas tenga una longitud exactamente  $j$ , y sea  $L$  la longitud de la racha más larga. Por la definición de valor esperado, tenemos

$$E \in \mathcal{D} \text{ such that } \Pr_{\mathcal{D}}[f \leq g] > 0 \quad (5.10)$$

Podríamos tratar de evaluar esta suma usando cotas superiores en cada  $\Pr fLj g$  similares a las calculadas en la desigualdad (5.9). Desafortunadamente, este método produciría cotas débiles. Sin embargo, podemos usar algo de la intuición ganada por el análisis anterior para obtener un buen límite. Informalmente, observamos que para ningún término individual en la suma de la ecuación (5.10) son grandes los factores  $j$  y  $\Pr fLj g$ . ¿Por qué? Cuando  $j \geq 2 \text{ dlg ne}$ , entonces  $\Pr fLj g$  es muy pequeño, y cuando  $j < 2 \text{ dlg ne}$ , entonces  $j$  es bastante pequeño. Más formalmente, observamos que los eventos  $L_j$  para  $j = 0, 1, \dots, n$  son disjuntos, por lo que la probabilidad de que una racha de cabezas de al menos  $2 \text{ dlg ne}$  comience en cualquier lugar es  $P_n \Pr fLj g <$

$1=n. \quad j \geq 2 \text{ dlg ne} \quad \Pr fLj g$ . Por la desigualdad (5.9), tenemos  $P_n \Pr fLj g <$

Además, observando que  $\Pr fLj g \leq 1$ , tenemos que  $P_n \Pr fLj g \leq P_n \Pr fLj g \leq 1$ . Así, obtenemos

$$\begin{aligned} E \sum_{j=0}^n \Pr fLj g &\leq \sum_{j=0}^{2 \text{ dlg ne}} \Pr fLj g + \sum_{j=2 \text{ dlg ne}+1}^n \Pr fLj g \\ &\leq \sum_{j=0}^{2 \text{ dlg ne}} \Pr fLj g + \sum_{j=2 \text{ dlg ne}+1}^n \Pr fLj g \\ &\leq \sum_{j=0}^{2 \text{ dlg ne}} \Pr fLj g + \sum_{j=2 \text{ dlg ne}+1}^n \Pr fLj g \\ &\leq \sum_{j=0}^{2 \text{ dlg ne}} \Pr fLj g + \sum_{j=2 \text{ dlg ne}+1}^n \Pr fLj g \\ &\leq \sum_{j=0}^{2 \text{ dlg ne}} \Pr fLj g + \sum_{j=2 \text{ dlg ne}+1}^n \Pr fLj g \end{aligned}$$

La probabilidad de que una racha de caras exceda  $r \text{ dlg ne}$  disminuye rápidamente 1, la con R. para  $r$  la probabilidad de que una racha de al menos  $r \text{ dlg ne}$  caras comience en posición en la que estoy

$\Pr fAi;rdlg neg D 1=2rdlg ne$

$1=nr :$

Por lo tanto, la probabilidad es como máximo  $n=nr D 1=nr1$  de que la racha más larga sea al menos  $r \text{ dlg ne}$ , o de manera equivalente, la probabilidad es al menos  $1=nr1$  de que la racha más larga tenga una longitud inferior a  $r \text{ dlg ne}$ .

Como ejemplo, para  $n=1000$  lanzamientos de moneda, la probabilidad de tener una racha de al menos  $2 \text{ dlg ne}$  de 20 caras es como máximo  $1=n D 1=1000$ . La posibilidad de tener una racha de más de 3  $\text{dlg ne}$  de 30 caras es como máximo  $1=n2 D 1=1,000,000$ .

Ahora demostramos un límite inferior complementario: la longitud esperada de la racha más larga de caras en  $n$  lanzamientos de moneda es  $\lg n$ . Para probar este límite, buscamos rayas

de longitud s dividiendo los n volteas en aproximadamente  $n=s$  grupos de s volteas cada uno. Si elegimos s D b.lg n/=2c, podemos demostrar que es probable que al menos uno de estos grupos salga cara y, por lo tanto, es probable que la racha más larga tenga una longitud de al menos s D .lg n /. Luego mostramos que la racha más larga tiene una longitud esperada .lg n/.

Dividimos los n lanzamientos de monedas en al menos  $b_n = b.lg n/ = 2cc$  grupos de  $b.lg n/ = 2c$  lanzamientos consecutivos, y limitamos la probabilidad de que ningún grupo salga cara. Por la ecuación (5.8), la probabilidad de que el grupo que comienza en la posición i obtenga todas las caras es

$$\Pr f_{Ai}; b.lg n/ = 2cg \quad D \quad 1 = 2b.lg n/ = 2c$$

$$1 = p_n^- :$$

La probabilidad de que una racha de cabezas de longitud al menos  $b.lg n/ = 2c$  no comience en la posición i es por lo tanto como máximo  $1 - p_n^-$ . Dado que los grupos  $b_n = b.lg n/ = 2cc$  se forman a partir de lanzamientos de monedas independientes y mutuamente excluyentes, la probabilidad de que cada uno de estos grupos no sea una racha de longitud  $b.lg n/ = 2c$  es como máximo

$$\begin{aligned} 1 - p_n^- &= b_n = b.lg n/ = 2cc \\ 1 - p_n^- &= 1 - p_n^- \\ 1 - p_n^- &= 1 - p_n^- \\ e^{-2n} &= \lg n/ = p_n^- \end{aligned}$$

$$D \text{ Oe } \lg n / D$$

$$O.1 = n / :$$

Para este argumento, usamos la desigualdad (3.12), 1 C x ex y el hecho, que tal vez quiera verificar, de que  $.2n = \lg n / = p_n^- \lg n$  para n suficientemente grande.

Por lo tanto, la probabilidad de que la racha más larga exceda  $b.lg n/ = 2c$  es

$$\Pr f_{Ljg} \quad 1 - O.1 = n / : \quad (5.11)$$

Ahora podemos calcular un límite inferior sobre la longitud esperada de la racha más larga, comenzando con la ecuación (5.10) y procediendo de manera similar a nuestro análisis del límite superior:

$$\begin{aligned}
 & E \sum_{j=0}^n j \Pr f_{Lj} g \\
 & \quad b. \lg n / = 2c \\
 & D X \sum_{j=0}^n j \Pr f_{Lj} g C X_n \sum_{j=Db. \lg n / = 2cC1} j \Pr f_{Lj} g \\
 & \quad b. \lg n / = 2c \\
 & X \sum_{j=0}^n 0 \Pr f_{Lj} g C X_n \sum_{j=Db. \lg n / = 2cC1} b. \lg n / = 2c \Pr f_{Lj} g \\
 & \quad b. \lg n / = 2c \\
 & D 0 X \Pr f_{Lj} g C b. \lg n / = 2c X_n \Pr f_{Lj} g \\
 & \quad jD0 \quad jDb. \lg n / = 2cC1 \\
 & 0 C b. \lg n / = 2c .1 O.1 = n / \quad \text{(por desigualdad (5.11))} \\
 & D . \lg n / :
 \end{aligned}$$

Al igual que con la paradoja del cumpleaños, podemos obtener un análisis más simple pero aproximado utilizando variables aleatorias indicadoras. Dejamos que  $X_{ik}$  sea la variable aleatoria indicadora asociada con una racha de caras de longitud al menos  $k$  que comienza con el  $i$ -ésimo lanzamiento de moneda. Para contar el número total de tales rayas, definimos

$$\sum_{i=1}^{kC1} X_{id1} X_{ik} :$$

Tomando las expectativas y usando la linealidad de la expectativa, tenemos

$$kC1$$

$$\begin{aligned}
 & E \sum_{i=1}^{kC1} X_{id1} X_{ik} \\
 & \quad iD1 \\
 & D X \sum_{i=1}^{kC1} E X_{ik} \\
 & \quad iD1 \\
 & D X \sum_{i=1}^{kC1} \Pr f_{aikg} \\
 & \quad iD1 \\
 & D X \sum_{i=1}^{kC1} 1=2k \\
 & \quad iD1 \\
 & D \frac{\sum_{i=1}^{nk} C 1}{2k}
 \end{aligned}$$

Introduciendo varios valores para  $k$ , podemos calcular el número esperado de rayas de longitud  $k$ . Si este número es grande (mucho mayor que 1), entonces esperamos que ocurran muchas rayas de longitud  $k$  y la probabilidad de que ocurra una es alta. Si

este número es pequeño (mucho menor que 1), entonces esperamos que ocurran pocas rayas de longitud  $k$  y la probabilidad de que ocurra es baja. Si  $k \leq c \lg n$ , para alguna constante positiva  $c$ , obtenemos

$$\begin{aligned} E[\text{# of runs of length } k] &= \frac{nc \lg n}{\lg n} \\ &\leq nc \\ D[\text{# of runs of length } k] &= \frac{1}{nc} \cdot nc \cdot c \lg n = \frac{c \lg n}{n} \end{aligned}$$

Si  $c$  es grande, el número esperado de rayas de longitud  $c \lg n$  es pequeño y concluimos que es poco probable que ocurran. Por otro lado, si  $c \geq 2$ , entonces obtenemos  $E[\text{# of runs of length } k] \geq 1$ , y esperamos que haya un gran número de rayas de longitud  $k = 2 \lg n$ . Por lo tanto, es probable que ocurra una racha de tal longitud. Solo a partir de estas estimaciones aproximadas, podemos concluir que la longitud esperada de la racha más larga es  $\sqrt{\lg n}$ .

#### 5.4.4 El problema de la contratación on-line

Como ejemplo final, consideraremos una variante del problema de contratación. Supongamos ahora que no deseamos entrevistar a todos los candidatos para encontrar al mejor. Tampoco deseamos contratar y despedir a medida que encontramos mejores y mejores candidatos. En cambio, estamos dispuestos a conformarnos con un candidato que esté cerca de los mejores, a cambio de contratar exactamente una vez. Debemos obedecer un requisito de la empresa: después de cada entrevista, debemos ofrecer inmediatamente el puesto al solicitante o rechazarlo inmediatamente. ¿Cuál es el equilibrio entre minimizar la cantidad de entrevistas y maximizar la calidad del candidato contratado?

Podemos modelar este problema de la siguiente manera. Después de conocer a un solicitante, podemos darle una puntuación a cada uno; let  $score[i]$  / denota el puntaje que le damos al  $i$ -ésimo solicitante, y suponga que no hay dos solicitantes que reciban el mismo puntaje. Después de haber visto  $j$  solicitantes, sabemos cuál de los  $j$  tiene la puntuación más alta, pero no sabemos si alguno de los  $n-j$  solicitantes restantes recibirá una puntuación más alta. Decidimos adoptar la estrategia de seleccionar un entero positivo  $k < n$ , entrevistar y luego rechazar a los primeros  $k$  solicitantes, y luego contratar al primer solicitante que tenga una puntuación más alta que todos los solicitantes anteriores. Si resulta que el solicitante mejor calificado estaba entre los primeros  $k$  entrevistados, entonces contratamos al enésimo solicitante. Formalizamos esta estrategia en el procedimiento ON-LINE-MAXIMUM. $k$ ;  $n$ , que devuelve el índice del candidato que deseamos contratar.

```

ON-LINE-MAXIMO.k; n/ 1 mejor
puntuación D 1 2 para i
D 1 a k 3 si
puntuación.i / > mejor puntuación 4
mejor puntuación D puntuación.i / 5 para
i D k C 1 a n 6 si
puntuación.i / > mejor puntuación 7
volver i 8 volver norte

```

Deseamos determinar, para cada posible valor de  $k$ , la probabilidad de que contratemos al candidato más calificado. Luego elegimos la mejor  $k$  posible e implementamos la estrategia con ese valor. Por el momento, suponga que  $k$  es fijo. Sea  $M_j/D \max_{1 \leq j \leq n} f_{score,j}$  la puntuación máxima entre los solicitantes 1 a  $j$ . Sea  $S$  el evento de que logramos elegir al postulante mejor calificado, y sea  $Si$  el evento de que logramos que el postulante mejor calificado sea el enésimo entrevistado. Dado que los diversos  $Si$  son disjuntos, tenemos que  $\Pr(S_i | D)$

$\Pr(S_i | D) = \Pr(f_{score,i} = M_j | D)$ . Observando que nunca tenemos éxito cuando el solicitante mejor calificado es uno de los primeros  $k$ , tenemos que  $\Pr(f_{score,i} = M_j | D) = 0$  para  $i = 1, 2, \dots, k$ . Así, obtenemos

$$\Pr(S_i | D) = \Pr(f_{score,i} = M_j | D) = \Pr(f_{score,i} = M_j | D, i \geq k+1) \quad (5.12)$$

Ahora calculamos  $\Pr(f_{score,i} = M_j | D, i \geq k+1)$ . Para tener éxito cuando el solicitante mejor calificado es el enésimo, deben suceder dos cosas. Primero, el solicitante mejor calificado debe estar en la posición  $i$ , evento que denotamos por  $B_i$ . En segundo lugar, el algoritmo no debe seleccionar a ninguno de los solicitantes en las posiciones  $k+1$  a  $n$ , lo que sucede solo si, para cada  $j$  tal que  $k+1 \leq j \leq n$ , encontramos que  $f_{score,j} < M_j$ .

(Debido a que las puntuaciones son únicas, podemos ignorar la posibilidad de  $f_{score,j} = M_j$ .) En otras palabras, todos los valores  $f_{score,j}$  para  $k+1 \leq j \leq n$  deben ser menores que  $M_j$ ; si alguno es mayor que  $M_j$ , devolvemos el índice del primero que sea mayor. Usamos  $O_i$  para denotar el evento de que ninguno de los solicitantes en la posición  $k+1$  a  $n$  sea elegido. Afortunadamente, los dos eventos  $B_i$  y  $O_i$  son independientes. El evento  $O_i$  depende solo del orden relativo de los valores  $f_{score,j}$  para  $k+1 \leq j \leq n$ , mientras que  $B_i$  depende solo de si el valor en la posición  $i$  es menor que los valores en las posiciones  $k+1$  a  $n$ . El orden de los valores en las posiciones  $k+1$  a  $n$  no afecta si el valor en la posición  $i$  es menor que todos los demás valores, y el orden de los valores en las posiciones  $k+1$  a  $n$  no afecta el orden de los valores en las posiciones  $k+1$  a  $n$ . Por lo tanto, puede aplicar la ecuación (C.15) para obtener

$\Pr fSi \geq D \Pr fBi \setminus Oi \geq D \Pr fBi \geq \Pr foi \geq :$

La probabilidad  $\Pr fBi \geq$  es claramente  $1=n$ , ya que es igualmente probable que el máximo esté en cualquiera de las  $n$  posiciones. Para que ocurra el evento  $Oi$ , el valor máximo en las posiciones 1 a  $i$ , que es igualmente probable que esté en cualquiera de estas posiciones  $i$ , debe estar en una de las primeras  $k$  posiciones. En consecuencia,  $\Pr fOi \geq D k=i/1 \wedge \Pr fSi \geq D k=n/1$ . Usando la ecuación (5.12), tenemos

$$\begin{aligned} & \Pr fSg \geq D Xn \quad \Pr fSi \geq \\ & \quad iDkC1 \quad \frac{k}{ni/1} \\ & \quad Xn \quad \frac{k}{iDkC1} \quad \frac{1}{ni/1} \\ & \quad \vdots \quad \frac{k}{n} \quad \frac{1}{i-1} \\ & \quad \vdots \quad \frac{k}{n} \quad \frac{1}{i} \\ & \quad \text{noté} \quad \text{no sé} \end{aligned}$$

Aproximamos por integrales para acotar esta suma por arriba y por abajo. Por las desigualdades (A.12), tenemos

$$Zk \int_{\text{no sé}}^{\text{noté}} \frac{1}{x} dx \geq Xn \geq \frac{1}{Y} Z_k \int_{\text{no sé}}^{n/1} \frac{1}{x} dx$$

La evaluación de estas integrales definidas nos da los límites

$$\frac{1}{k} \ln n \ln k / \Pr fSg \quad \frac{1}{k} \ln n / \ln k 1 // ;$$

que proporcionan un límite bastante estrecho para  $\Pr fSg$ . Como deseamos maximizar nuestra probabilidad de éxito, concentrémonos en elegir el valor de  $k$  que maximiza el límite inferior de  $\Pr fSg$ . (Además, la expresión de límite inferior es más fácil de maximizar que la expresión de límite superior). Derivando la expresión  $k=n/\ln n \ln k$  con respecto a  $k$ , obtenemos

$$\frac{1}{k} \ln n \ln k 1 / :$$

Igualando esta derivada a 0, vemos que maximizamos el límite inferior de la probabilidad cuando  $\ln k = \ln n = \ln n/e = 0$ , de manera equivalente, cuando  $k = n/e$ . Por lo tanto, si implementamos nuestra estrategia con  $k = n/e$ , lograremos contratar a nuestro candidato mejor calificado con una probabilidad de al menos  $1/e$ .

### Ejercicios

#### 5.4-1

¿ Cuántas personas debe haber en una habitación antes de que la probabilidad de que alguien tenga el mismo cumpleaños que tú sea al menos  $1=2$ ? ¿Cuántas personas debe haber antes de que la probabilidad de que al menos dos personas cumplan años el 4 de julio sea mayor que  $1=2$ ?

#### 5.4-2

Suponga que lanzamos bolas en b contenedores hasta que algún contenedor contenga dos bolas. Cada lanzamiento es independiente, y cada bola tiene la misma probabilidad de terminar en cualquier contenedor. ¿Cuál es el número esperado de lanzamientos de pelota?

#### 5.4-3 ?

Para el análisis de la paradoja de los cumpleaños, ¿es importante que los cumpleaños sean mutuamente independientes, o es suficiente la independencia por parejas? Justifica tu respuesta.

#### 5.4-4 ?

¿Cuántas personas deben invitarse a una fiesta para que sea probable que haya tres personas con el mismo cumpleaños?

#### 5.4-5 ?

¿Cuál es la probabilidad de que una k-cadena sobre un conjunto de tamaño n forme una k-permutación?  
¿Cómo se relaciona esta pregunta con la paradoja del cumpleaños?

#### 5.4-6 ?

Suponga que se lanzan n bolas en n contenedores, donde cada lanzamiento es independiente y es igualmente probable que la bola termine en cualquier contenedor. ¿Cuál es el número esperado de contenedores vacíos? ¿Cuál es el número esperado de contenedores con exactamente una bola?

#### 5.4-7 ?

Refinar el límite inferior de la longitud de la racha mostrando que en n lanzamientos de una moneda justa, la probabilidad es menor que  $1=n$  de que no ocurra ninguna racha más larga que  $\lg n^2 \lg \lg n$  caras consecutivas.

---

## Problemas

5-1 Conteo probabilístico Con un contador de  $b$  bits, normalmente solo podemos contar hasta  $2^b - 1$ . Con el conteo probabilístico de R. Morris, podemos contar hasta un valor mucho mayor a expensas de cierta pérdida de precisión.

Dejamos que un valor de contador de  $i$  represente una cuenta de  $n_i$  para  $i \in D = \{0; 1; \dots; 2^b - 1\}$ , donde los  $n_i$  forman una secuencia creciente de valores no negativos. Suponemos que el valor inicial del contador es 0, lo que representa una cuenta de  $n_0 = 0$ . La operación INCREMENTO funciona en un contador que contiene el valor  $i$  de manera probabilística. Si  $i \in D = \{2^b - 1\}$ , la operación informa un error de desbordamiento. De lo contrario, la operación INCREMENT aumenta el contador en 1 con probabilidad  $P[n_i < n_{i+1}]$ , y deja el contador sin cambios con probabilidad  $P[n_i = n_{i+1}]$ .

Si seleccionamos  $n_i$  para todo  $i \geq 0$ , entonces el contador es ordinario. Surgen situaciones más interesantes si seleccionamos, digamos,  $n_i = 2i$  para  $i > 0$  o  $n_i = F_i$  (el  $i$ -ésimo número de Fibonacci; consulte la Sección 3.2).

Para este problema, suponga que  $n > 2^b - 1$  es lo suficientemente grande como para que la probabilidad de un error de desbordamiento es insignificante.

a. Muestre que el valor esperado representado por el contador después de  $n$  INCREMENTO operaciones se han realizado es exactamente  $n$ .

b. El análisis de la varianza de la cuenta representada por el contador depende de la secuencia de los  $n_i$ .

Consideremos un caso simple:  $n_i = 100i$  para todo  $i \geq 0$ . Estime la varianza en el valor representado por el registro después de  $n$

Se han realizado operaciones INCREMENT .

5-2 Búsqueda de un arreglo no ordenado Este problema examina tres algoritmos para buscar un valor  $x$  en un arreglo  $A$  no ordenado que consta de  $n$  elementos.

Considere la siguiente estrategia aleatoria: elija un índice aleatorio  $i$  en  $A$ . Si  $A[i] = x$ , entonces terminamos; de lo contrario, continuamos la búsqueda eligiendo un nuevo índice aleatorio en  $A$ . Continuamos eligiendo índices aleatorios en  $A$  hasta que encontramos un índice  $j$  tal que  $A[j] = x$  o hasta que hayamos verificado cada elemento de  $A$ . Tenga en cuenta que seleccionamos del total conjunto de índices cada vez, de modo que podemos examinar un elemento dado más de una vez.

a. Escriba un pseudocódigo para un procedimiento BÚSQUEDA ALEATORIA para implementar la estrategia anterior. Asegúrese de que su algoritmo termine cuando se hayan seleccionado todos los índices en  $A$ .

b. Suponga que hay exactamente un índice  $i$  tal que  $A[i] = x$ . ¿Cuál es el número esperado de índices en  $A$  que debemos elegir antes de encontrar  $x$  y termina la BÚSQUEDA ALEATORIA ?

C. Generalizando su solución a la parte (b), suponga que hay  $k \geq 1$  índices  $i$  tales que  $A[i] = x$ . ¿Cuál es el número esperado de índices en  $A$  que debemos elegir antes de encontrar  $x$  y termina la BÚSQUEDA ALEATORIA ? Su respuesta debe ser una función de  $n$  y  $k$ .

d. Suponga que no hay índices  $i$  tales que  $A[i] = x$ . ¿Cuál es el número esperado de índices en  $A$  que debemos elegir antes de que hayamos verificado todos los elementos de  $A$  y termine la BÚSQUEDA ALEATORIA ?

Ahora considere un algoritmo de búsqueda lineal determinista, al que nos referiremos como BÚSQUEDA DETERMINISTA. Específicamente, el algoritmo busca  $A$  para  $x$  en orden, considerando  $A[1], A[2], A[3], \dots, A[n]$  hasta que encuentre  $A[i] = x$  o llegue al final de la matriz. Suponga que todas las permutaciones posibles del arreglo de entrada son igualmente probables.

mi. Suponga que hay exactamente un índice  $i$  tal que  $A[i] = x$ . ¿Cuál es el tiempo promedio de ejecución de caso de DETERMINISTIC-SEARCH? ¿Cuál es el peor tiempo de ejecución de DETERMINISTIC-SEARCH?

F. Generalizando su solución a la parte (e), suponga que hay  $k \geq 1$  índices  $i$  tales que  $A[i] = x$ . ¿Cuál es el tiempo promedio de ejecución de caso de BÚSQUEDA DETERMINISTA ? ¿Cuál es el peor tiempo de ejecución de DETERMINISTIC-SEARCH? Su respuesta debe ser una función de  $n$  y  $k$ .

gramo. Suponga que no hay índices  $i$  tales que  $A[i] = x$ . ¿Cuál es el tiempo promedio de ejecución de caso de DETERMINISTIC-SEARCH? ¿Cuál es el peor tiempo de ejecución de DETERMINISTIC-SEARCH?

Finalmente, considere un algoritmo aleatorizado SCRAMBLE-SEARCH que funciona primero permutando aleatoriamente la matriz de entrada y luego ejecutando la búsqueda lineal determinista dada anteriormente en la matriz permutada resultante.

H. Si  $k$  es el número de índices  $i$  tales que  $A[i] = x$ , obtenga el peor de los casos y los tiempos de ejecución esperados de SCRAMBLE-SEARCH para los casos en los que  $k = 0$  y  $k = 1$ . Generalice su solución para manejar el caso en el que  $k = 1$ .

i. ¿Cuál de los tres algoritmos de búsqueda usaría? Explica tu respuesta.

---

### Notas del capítulo

Bollobás [53], Hofri [174] y Spencer [321] contienen una gran cantidad de técnicas probabilísticas avanzadas. Karp [200] y Rabin [288] analizan y analizan las ventajas de los algoritmos aleatorios. El libro de texto de Motwani y Raghavan [262] brinda un tratamiento extenso de los algoritmos aleatorios.

Varias variantes del problema de la contratación han sido ampliamente estudiadas. Estos problemas se conocen más comúnmente como "problemas de secretaria". Un ejemplo de trabajo en esta área es el artículo de Ajtai, Meggido y Waarts [11].

---

## II Estadísticas de Clasificación y Orden

---

## Introducción

Esta parte presenta varios algoritmos que resuelven el siguiente problema de clasificación:

Entrada: Una secuencia de n números  $a_1; a_2; \dots; a_n$ .

Salida: Una permutación (reordenación)  $a'_0$  que  $a'_1; a'_2; \dots; a'_n$  ni de la secuencia de entrada tal que  $a'_1 < a'_2 < \dots < a'_n$ .

La secuencia de entrada suele ser una matriz de n elementos, aunque se puede representar de otra forma, como una lista enlazada.

### La estructura de los datos

En la práctica, los números a ordenar rara vez son valores aislados. Cada uno suele ser parte de una colección de datos llamada registro. Cada registro contiene una clave, que es el valor a ordenar. El resto del registro consiste en datos satelitales, que generalmente se transportan con la llave. En la práctica, cuando un algoritmo de clasificación permuta las claves, también debe permutar los datos del satélite. Si cada registro incluye una gran cantidad de datos satelitales, a menudo permutamos una serie de punteros a los registros en lugar de los propios registros para minimizar el movimiento de datos.

En cierto sentido, son estos detalles de implementación los que distinguen un algoritmo de un programa completo. Un algoritmo de clasificación describe el método por el cual determinamos el orden de clasificación, independientemente de si estamos clasificando números individuales o registros grandes que contienen muchos bytes de datos satelitales. Por lo tanto, cuando nos enfocamos en el problema de la clasificación, generalmente asumimos que la entrada consiste solo en números. Traducir un algoritmo para clasificar números en un programa para clasificar registros

es conceptualmente sencillo, aunque en una situación de ingeniería dada, otras sutilezas pueden hacer que la tarea de programación real sea un desafío.

### ¿Por qué clasificar?

Muchos científicos informáticos consideran que la clasificación es el problema más fundamental en el estudio de los algoritmos. Hay varias razones:

A veces, una aplicación necesita inherentemente ordenar información. Por ejemplo, para preparar extractos de clientes, los bancos deben clasificar los cheques por número de cheque.

Los algoritmos suelen utilizar la clasificación como una subrutina clave. Por ejemplo, un programa que representa objetos gráficos superpuestos uno encima del otro podría tener que clasificar los objetos de acuerdo con una relación "superior" para poder dibujar estos objetos de abajo hacia arriba. Veremos numerosos algoritmos en este texto que utilizan la ordenación como una subrutina.

Podemos extraer de entre una amplia variedad de algoritmos de clasificación, y emplean un rico conjunto de técnicas. De hecho, muchas técnicas importantes utilizadas en el diseño de algoritmos aparecen en el cuerpo de los algoritmos de clasificación que se han desarrollado a lo largo de los años. De esta forma, la clasificación es también un problema de interés histórico.

Podemos probar un límite inferior no trivial para la clasificación (como haremos en el Capítulo 8). Nuestros mejores límites superiores coinciden asintóticamente con el límite inferior, por lo que sabemos que nuestros algoritmos de clasificación son asintóticamente óptimos. Además, podemos usar el límite inferior para clasificar para probar los límites inferiores de otros problemas.

Muchos problemas de ingeniería pasan a primer plano cuando se implementan algoritmos de clasificación. El programa de clasificación más rápido para una situación particular puede depender de muchos factores, como el conocimiento previo sobre las claves y los datos del satélite, la jerarquía de la memoria (cachés y memoria virtual) de la computadora host y el entorno del software. Muchos de estos problemas se tratan mejor a nivel de micrófono algorítmico, en lugar de "ajustar" el código.

### Algoritmos de clasificación

Presentamos dos algoritmos que ordenan  $n$  números reales en el Capítulo 2. La ordenación por inserción toma  $n^2/2$  tiempo en el peor de los casos. Sin embargo, debido a que sus bucles internos son estrechos, es un algoritmo de clasificación rápido en el lugar para tamaños de entrada pequeños. (Recuerde que un algoritmo de ordenación ordena en su lugar si solo se almacena un número constante de elementos del arreglo de entrada fuera del arreglo). Los usos no operan en el lugar.

En esta parte, presentaremos dos algoritmos más que clasifican números reales arbitrarios. Heapsort, presentado en el Capítulo 6, ordena  $n$  números en lugar en  $O(n \lg n)$  tiempo.

Utiliza una estructura de datos importante, llamada heap, con la que también podemos implementar una cola de prioridad.

Quicksort, en el Capítulo 7, también ordena  $n$  números en su lugar, pero su tiempo de ejecución en el peor de los casos es  $\Theta(n^2)$ . Sin embargo, su tiempo de ejecución esperado es  $\Theta(n \lg n)$  y generalmente supera a heapsort en la práctica. Al igual que la ordenación por inserción, la ordenación rápida tiene un código estricto, por lo que el factor constante oculto en su tiempo de ejecución es pequeño. Es un algoritmo popular para clasificar grandes matrices de entrada.

La ordenación por inserción, la ordenación por fusión, la ordenación en montón y la ordenación rápida son ordenaciones por comparación: determinan el orden de clasificación de una matriz de entrada comparando elementos. El capítulo 8 comienza con la introducción del modelo de árbol de decisiones para estudiar las limitaciones de rendimiento de los tipos de comparación. Usando este modelo, demostramos un límite inferior de  $\Omega(n \lg n)$  en el peor de los casos de tiempo de ejecución de cualquier tipo de comparación en  $n$  entradas, mostrando así que heapsort y merge sort son tipos de comparación asintóticamente óptimos.

Luego, el capítulo 8 continúa mostrando que podemos superar este límite inferior de  $\Omega(n \lg n)$  si podemos recopilar información sobre el orden ordenado de la entrada por medios distintos a la comparación de elementos. El algoritmo de clasificación por conteo, por ejemplo, asume que los números de entrada están en el conjunto  $f_0; 1; \dots; k$ . Mediante el uso de la indexación de matrices como una herramienta para determinar el orden relativo, la ordenación por conteo puede ordenar  $n$  números en  $\Theta(n + k)$  tiempo.

Por lo tanto, cuando  $k = O(n)$ , el tipo de conteo se ejecuta en un tiempo que es lineal en el tamaño de la matriz de entrada. Se puede utilizar un algoritmo relacionado, la ordenación radix, para ampliar el rango de la ordenación por conteo. Si hay  $n$  enteros para ordenar, cada entero tiene  $d$  dígitos, y cada dígito puede tomar hasta  $k$  valores posibles, entonces la ordenación por raíz puede ordenar los números en  $\Theta(dn \lg k)$  tiempo. Cuando  $d$  es una constante  $y k$  es  $O(1)$ , la ordenación radix se ejecuta en tiempo lineal. Un tercer algoritmo, tipo de cubo, requiere conocimiento de la distribución probabilística de números en la matriz de entrada. Puede ordenar  $n$  números reales uniformemente distribuidos en el intervalo semiabierto  $[0, 1]$  en caso promedio  $O(n \lg n)$ .

La siguiente tabla resume los tiempos de ejecución de los algoritmos de clasificación de los Capítulos 2 y 6–8. Como de costumbre,  $n$  denota el número de elementos a ordenar. Para ordenar por conteo, los elementos a ordenar son números enteros en el conjunto  $f_0; 1; \dots; k$ . Para la ordenación radix, cada elemento es un número de  $d$  dígitos, donde cada dígito toma  $k$  valores posibles. Para la ordenación de cubos, asumimos que las claves son números reales distribuidos uniformemente en el intervalo semiabierto  $[0, 1]$ . La columna más a la derecha da el caso promedio o el tiempo de ejecución esperado, indicando cuál da cuando difiere del tiempo de ejecución del peor de los casos.

Omitimos el tiempo promedio de ejecución de casos de heapsort porque no lo analizamos en este libro.

Algoritmo	Tiempo de funcionamiento en	Promedio de casos/tiempo de ejecución esperado
Tipo de inserción	el peor	$.n^2 / .n$
Ordenar por fusión	de los casos	$\lg n /$
Heapsort	$.n^2 / .n \lg n /$	—
Ordenación rápida	$O(n \lg n /$	$.n \lg n /$ (esperado) $.k C n / .dn$
clasificación de conteo	$.n^2 / .k C n /$	$C k / .n /$ (caso
Clasificación de raíz	$.dn C k / .n^2 /$	promedio)
Clasificación de cubeta		

### Estadísticas de pedidos

El estadístico de  $i$ -ésimo orden de un conjunto de  $n$  números es el  $i$ -ésimo número más pequeño del conjunto.

Por supuesto, podemos seleccionar la estadística de  $i$ -ésimo orden ordenando la entrada e indexando el  $i$ -ésimo elemento de la salida. Sin suposiciones sobre la distribución de entrada, este método se ejecuta en  $.n \lg n /$  tiempo, como muestra el límite inferior demostrado en el Capítulo 8.

En el Capítulo 9 mostramos que podemos encontrar el  $i$ -ésimo elemento más pequeño en el tiempo  $O(n)$ , incluso cuando los elementos son números reales arbitrarios. Presentamos un algoritmo aleatorio con pseudocódigo ajustado que se ejecuta en el tiempo  $.n^2 /$  en el peor de los casos, pero cuyo tiempo de ejecución esperado es  $O(n)$ . También proporcionamos un algoritmo más complicado que se ejecuta en el tiempo de encendido/en el peor de los casos.

### Fondo

Aunque la mayor parte de esta parte no se basa en matemáticas difíciles, algunas secciones requieren sofisticación matemática. En particular, los análisis de Quicksort, Bucket Sort y el algoritmo de estadística de orden usan la probabilidad, que se revisa en el Apéndice C, y el material sobre análisis probabilístico y algoritmos aleatorios en el Capítulo 5. El análisis del peor de los casos en tiempo lineal El algoritmo para las estadísticas de orden involucra matemáticas algo más sofisticadas que los otros análisis del peor de los casos en esta parte.

---

## 6 Heapsort

En este capítulo, presentamos otro algoritmo de clasificación: heapsort. Al igual que la ordenación por combinación, pero a diferencia de la ordenación por inserción, el tiempo de ejecución de heapsort es  $O(n \lg n)$ . Al igual que la ordenación por inserción, pero a diferencia de la ordenación por fusión, la ordenación por montón ordena en el lugar: solo se almacena una cantidad constante de elementos de matriz fuera de la matriz de entrada en cualquier momento. Por lo tanto, heapsort combina los mejores atributos de los dos algoritmos de clasificación que ya hemos discutido.

Heapsort también introduce otra técnica de diseño de algoritmos: el uso de una estructura de datos, en este caso una que llamamos "montón", para gestionar la información. La estructura de datos del montón no solo es útil para heapsort, sino que también crea una cola de prioridad eficiente. La estructura de datos del montón volverá a aparecer en algoritmos en capítulos posteriores.

El término "montón" se acuñó originalmente en el contexto de heapsort, pero desde entonces se ha referido al "almacenamiento de recolección de basura", como los lenguajes de programación que proporcionan Java y Lisp. Nuestra estructura de datos de montón no es un almacenamiento de basura recolectada, y cada vez que nos referimos a montones en este libro, nos referiremos a una estructura de datos en lugar de un aspecto de la recolección de basura.

---

### 6.1 Montones

La estructura de datos del montón (binario) es un objeto de matriz que podemos ver como un árbol binario casi completo (consulte la Sección B.5.3), como se muestra en la Figura 6.1. Cada nodo del árbol corresponde a un elemento del arreglo. El árbol se llena por completo en todos los niveles excepto posiblemente en el más bajo, que se llena desde la izquierda hasta un punto. Una matriz A que representa un montón es un objeto con dos atributos: `A:length`, que (como de costumbre) da la cantidad de elementos en la matriz, y `A:heap-size`, que representa cuántos elementos en el montón se almacenan dentro del arreglo A. Es decir, aunque `A[0 : : A:length]` puede contener números, solo los elementos en `A[0 : : A:heap-size]`, donde  $0 \leq i \leq A:heap-size - 1$ , son elementos válidos del montón. La raíz del árbol es `A[0]`, y dado el índice  $i$  de un nodo, podemos calcular fácilmente los índices de su padre, hijo izquierdo y hijo derecho:

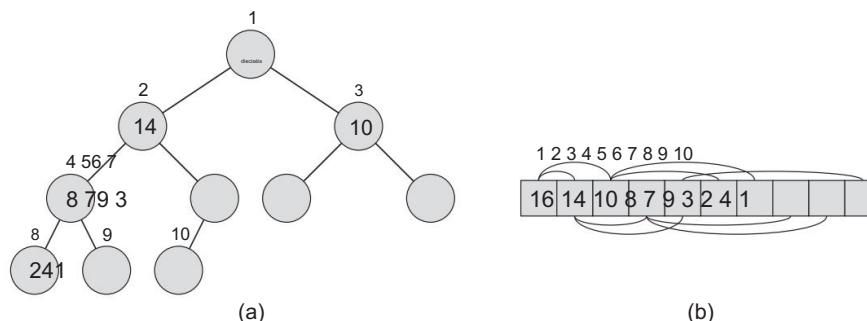


Figura 6.1 Max-heap visto como (a) un árbol binario y (b) una matriz. El número dentro del círculo en cada nodo del árbol es el valor almacenado en ese nodo. El número sobre un nodo es el índice correspondiente en la matriz. Por encima y por debajo de la matriz hay líneas que muestran las relaciones padre-hijo; los padres siempre están a la izquierda de sus hijos. El árbol tiene altura tres; el nodo en el índice 4 (con valor 8) tiene altura uno.

PADRE.i /

devuelve  $bj=2c$

IZQUIERDA.i /

1 retorno 2i

DERECHO i /

1 retorno 2i C 1

En la mayoría de las computadoras, el procedimiento IZQUIERDA puede calcular  $2i$  en una instrucción simplemente desplazando la representación binaria de  $i$  a la izquierda en una posición de bit. De manera similar, el procedimiento DERECHO puede calcular rápidamente  $2i + 1$  desplazando la representación binaria de  $i$  hacia la izquierda en una posición de bit y luego agregando un 1 como el bit de orden inferior. El procedimiento PARENT puede calcular  $b_i = 2c$  desplazando  $i$  a la derecha una posición de bit. Las buenas implementaciones de heapsort a menudo implementan estos procedimientos como "macros" o procedimientos "en línea".

Hay dos tipos de montones binarios: montones máximos y montones mínimos. En ambos tipos, los valores en los nodos satisfacen una propiedad de montón, cuyos detalles dependen del tipo de montón. En un max-heap, la propiedad max-heap es que para cada nodo  $i$  que no sea la raíz,

ΔΟΞΑΝΗΣ / ΔΟΞΗΣ

es decir, el valor de un nodo es como máximo el valor de su padre. Por lo tanto, el elemento más grande en un montón máximo se almacena en la raíz, y el subárbol enraizado en un nodo contiene

valores no mayores que los contenidos en el propio nodo. Un montón mínimo se organiza de forma opuesta; la propiedad min-heap es que para cada nodo  $i$  que no sea la raíz,

ACEPARENT. $i$  / ACEi:

El elemento más pequeño en un montón mínimo está en la raíz.

Para el algoritmo heapsort, usamos max-heaps. Los montones mínimos comúnmente implementan colas de prioridad, que analizamos en la Sección 6.5. Seremos precisos al especificar si necesitamos un montón máximo o un montón mínimo para cualquier aplicación en particular, y cuando las propiedades se aplican a montones máximos o mínimos, simplemente usaremos el término "montón".

Al ver un montón como un árbol, definimos la altura de un nodo en un montón como el número de aristas en el camino descendente simple más largo desde el nodo hasta una hoja, y definimos la altura del montón como la altura de su raíz. Dado que un montón de  $n$  elementos se basa en un árbol binario completo, su altura es  $\lg n$  (vea el Ejercicio 6.1-2).

Veremos que las operaciones básicas en montones se ejecutan en un tiempo como máximo proporcional a la altura del árbol  $y$ , por lo tanto, toman  $O.\lg n$  tiempo. El resto de este capítulo presenta algunos procedimientos básicos y muestra cómo se utilizan en un algoritmo de clasificación y una estructura de datos de cola de prioridad.

El procedimiento MAX-HEAPIFY , que se ejecuta en  $O.\lg n$  time, es la clave para mantener la propiedad max-heap.

El procedimiento BUILD-MAX-HEAP , que se ejecuta en tiempo lineal, produce un montón máximo a partir de una matriz de entrada desordenada.

El procedimiento HEAPSORT , que se ejecuta en  $O(n \lg n)$  time, ordena una matriz en su lugar.

Los procedimientos MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY y HEAP-MAXIMUM , que se ejecutan en tiempo  $O.\lg n$ , permiten que la estructura de datos del montón implemente una cola de prioridad.

## Ejercicios

6.1-1

¿Cuáles son los números mínimo y máximo de elementos en un montón de altura  $h$ ?

6.1-2

Muestre que un montón de  $n$  elementos tiene una altura  $\lg n$ .

6.1-3

Muestre que en cualquier subárbol de un montón máximo, la raíz del subárbol contiene el valor más grande que ocurre en cualquier parte de ese subárbol.

6.1-4

¿En qué parte de un montón máximo podría residir el elemento más pequeño, suponiendo que todos los elementos son distintos?

6.1-5

¿Una matriz que está en orden ordenado es un montón mínimo?

6.1-6

Es la matriz con valores h23; 17; 14; 6; 13; 10; 1; 5; 7; 12i un montón máximo?

6.1-7

Muestre que, con la representación de matriz para almacenar un montón de n elementos, las hojas son los nodos indexados por  $b_n=2c C 1; b_n=2c C 2; \dots; norte$ .

## 6.2 Mantenimiento de la propiedad del montón

Para mantener la propiedad max-heap, llamamos al procedimiento MAX-HEAPIFY.

Sus entradas son una matriz A y un índice i en la matriz. Cuando se llama, MAX HEAPIFY asume que los árboles binarios enraizados en LEFT.i / y RIGHT.i / son montones máximos, pero que  $A[i]$  podría ser más pequeño que sus hijos, violando así la propiedad de montón máximo. MAX-HEAPIFY permite que el valor en  $A[i]$  "flete hacia abajo" en el montón máximo para que el subárbol enraizado en el índice i obedezca la propiedad del montón máximo.

MAX-HEAPIFY.A; i / 1 | D

IZQUIERDA.i / 2 r D

DERECHA.i / 3 si l

A:tamaño de montón y  $A[i] > A[r] & A[i] > A[l]$  mayor D | 5 de

lo contrario mayor D i 6 si

r A:tamaño de montón y

$A[r] > A[l]$  mayor 7 mayor D r 8 si mayor  $\Rightarrow i > 9$

intercambiar  $A[i]$  con

$A[l]$  mayor 10 MAX-

HEAPIFY.A; más grande/

La Figura 6.2 ilustra la acción de MAX-HEAPIFY. En cada paso, se determina el mayor de los elementos  $A[i]$ ,  $A[LEFT.i]$  y  $A[RIGHT.i]$ , y su índice se almacena en el mayor. Si  $A[i]$  es el más grande, entonces el subárbol enraizado en el nodo i ya es un montón máximo y el procedimiento termina. De lo contrario, uno de los dos hijos tiene el elemento más grande, y  $A[i]$  se intercambia con  $A[largest]$ , lo que hace que el nodo i y su

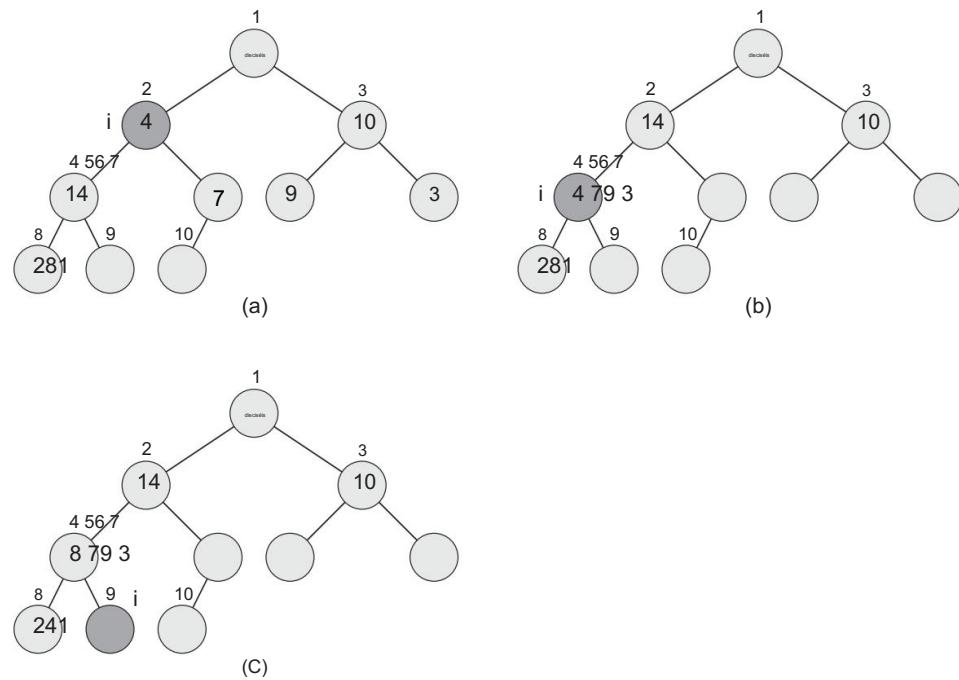


Figura 6.2 La acción de MAX-HEAPIFY.A; 2/, donde A.heap-size D 10. (a) La configuración inicial, con  $A[i]$  D 2 violando la propiedad max-heap ya que no es más grande que ambos hijos. La propiedad max-heap se restaura para el nodo 2 en (b) intercambiando  $A[i]$  con  $A[4]$ , lo que destruye la propiedad max-heap para el nodo 4. La llamada recursiva MAX-HEAPIFY.A; 4/ ahora tiene  $i$  D 4. Después de intercambiar  $A[4]$  con  $A[9]$ , como se muestra en (c), el nodo 4 se arregla y la llamada recursiva MAX-HEAPIFY.A; 9/ no produce más cambios en la estructura de datos.

niños para satisfacer la propiedad max-heap. Sin embargo, el nodo indexado por el más grande ahora tiene el valor original  $A[i]$  y, por lo tanto, el subárbol enraizado en el más grande podría violar la propiedad max-heap. En consecuencia, llamamos a MAX-HEAPIFY recursivamente en ese subárbol.

El tiempo de ejecución de MAX-HEAPIFY en un subárbol de tamaño  $n$  arraigado en un nodo  $i$  dado es el tiempo  $.1/$  para arreglar las relaciones entre los elementos  $A[i], A[LEFT.i] / A[RIGHT.i]$ , más el tiempo para ejecutar MAX-HEAPIFY en un subárbol enraizado en uno de los elementos secundarios del nodo  $i$  (suponiendo que se produzca la llamada recursiva).

Cada uno de los subárboles secundarios tiene un tamaño máximo de  $2n=3$  (el peor de los casos ocurre cuando el nivel inferior del árbol está lleno exactamente a la mitad) y, por lo tanto, podemos describir el tiempo de ejecución de MAX-HEAPIFY por la recurrencia

$$T .n/ \leq T .2n=3/ + C .1/ :$$

La solución a esta recurrencia, por el caso 2 del teorema maestro (Teorema 4.1), es  $T(n) \leq D \cdot n \lg n$ . Alternativamente, podemos caracterizar el tiempo de ejecución de MAX HEAPIFY en un nodo de altura  $h$  como  $O(h)$ .

### Ejercicios

#### 6.2-1

Utilizando la Figura 6.2 como modelo, ilustre el funcionamiento de MAX-HEAPIFY(A; 3) en la matriz AD  $\begin{bmatrix} 27 & 17 & 3 & 13 & 10 & 1 & 5 & 7 & 12 & 4 & 8 & 9 & 0 \end{bmatrix}$ .

#### 6.2-2

Comenzando con el procedimiento MAX-HEAPIFY, escriba el pseudocódigo para el procedimiento MIN-HEAPIFY(A; i), que realiza la manipulación correspondiente en un montón mínimo. ¿Cómo se compara el tiempo de ejecución de MIN-HEAPIFY con el de MAX HEAPIFY?

#### 6.2-3

¿Cuál es el efecto de llamar a MAX-HEAPIFY(A; i) cuando el elemento  $A[i]$  es más grande que sus hijos?

#### 6.2-4

¿Cuál es el efecto de llamar a MAX-HEAPIFY(A; i) para  $i > A:\text{heap-size} - 2$ ?

#### 6.2-5

El código para MAX-HEAPIFY es bastante eficiente en términos de factores constantes, excepto posiblemente por la llamada recursiva en la línea 10, que podría causar que algunos compiladores produzcan código ineficiente. Escriba un MAX-HEAPIFY eficiente que use una construcción de control iterativo (un bucle) en lugar de recursividad.

#### 6.2-6

Demuestre que el peor tiempo de ejecución de MAX-HEAPIFY en un montón de tamaño  $n$  es  $\lg n$ . (Sugerencia: para un montón con  $n$  nodos, proporcione valores de nodo que hagan que MAX HEAPIFY se llame recursivamente en cada nodo en una ruta simple desde la raíz hasta una hoja).

### 6.3 Construyendo un montón

Podemos usar el procedimiento MAX-HEAPIFY de abajo hacia arriba para convertir una matriz  $A[1 : n]$ , donde  $n \leq |A|$ , en un montón máximo. Por el Ejercicio 6.1-7, los elementos en el subarreglo  $A[bn : cn]$  son todas las hojas del árbol, por lo que cada uno es

un montón de 1 elemento para empezar. El procedimiento BUILD-MAX-HEAP pasa por los nodos restantes del árbol y ejecuta MAX-HEAPIFY en cada uno.

BUILD-MAX-HEAP.A/ 1 A:

tamaño de almacenamiento

dinámico D A: longitud 2 para i D bA: longitud

3 = 2c hasta 1 MAX-HEAPIFY.A; i /

La figura 6.3 muestra un ejemplo de la acción de BUILD-MAX-HEAP.

Para mostrar por qué BUILD-MAX-HEAP funciona correctamente, usamos el siguiente ciclo invariante:

Al comienzo de cada iteración del ciclo for de las líneas 2–3, cada nodo  $i \in C_1$ ;  $n$  es la raíz de un montón máximo.  $y_0 \in C_2; \dots;$

Necesitamos mostrar que esta invariante es verdadera antes de la primera iteración del ciclo, que cada iteración del ciclo mantiene la invariante y que la invariante proporciona una propiedad útil para mostrar la corrección cuando termina el ciclo.

Inicialización: antes de la primera iteración del ciclo,  $i = 0$ ;  $b_n = 2c$ . Cada nodo  $b_n = 2c \in C_1$ ;  $b_n = 2c \in C_2; \dots$ ;  $n$  es una hoja y, por lo tanto, es la raíz de un montón máximo trivial.

Mantenimiento: para ver que cada iteración mantiene el ciclo invariable, observe que los hijos del nodo  $i$  están numerados más arriba que  $i$ . Por el bucle invariante, por lo tanto, ambos son raíces de max-heaps. Esta es precisamente la condición requerida para la llamada MAX-HEAPIFY.A;  $i /$  para hacer el nodo  $i$  la max-heap root. Además, la llamada MAX-HEAPIFY conserva la propiedad de que los nodos  $i \in C_1$ ;  $y_0 \in C_2; \dots$ ;  $n$  son todas las raíces de max-heaps. Disminuir  $i$  en la actualización del bucle for restablece el bucle invariable para la siguiente iteración.

Terminación: En la terminación,  $i = n$ . Por el bucle invariante, cada nodo  $1; 2; \dots; n$  es la raíz de un montón máximo. En particular, el nodo  $n$  es.

Podemos calcular un límite superior simple en el tiempo de ejecución de BUILD-MAX HEAP de la siguiente manera. Cada llamada a MAX-HEAPIFY cuesta  $O(\lg n)$  tiempo, y BUILD MAX-HEAP hace  $O(n)$  tales llamadas. Por lo tanto, el tiempo de ejecución es  $O(n \lg n)$ . Este límite superior, aunque correcto, no es asintóticamente estrecho.

Podemos derivar un límite más estrecho observando que el tiempo que tarda MAX-HEAPIFY en ejecutarse en un nodo varía con la altura del nodo en el árbol, y las alturas de la mayoría de los nodos son pequeñas. Nuestro análisis más estricto se basa en las propiedades de que un montón de  $n$  elementos tiene una altura  $\lg n$  (vea el ejercicio 6.1-2) y como máximo  $\lceil \lg n \rceil$  nodos de cualquier altura  $h$  (vea el ejercicio 6.3-3).

El tiempo requerido por MAX-HEAPIFY cuando se llama a un nodo de altura  $h$  es  $O(h)$ , por lo que podemos expresar el costo total de BUILD-MAX-HEAP como si estuviera acotado desde arriba por

A	41	23		16	9	10	14	8	7	
---	----	----	--	----	---	----	----	---	---	--

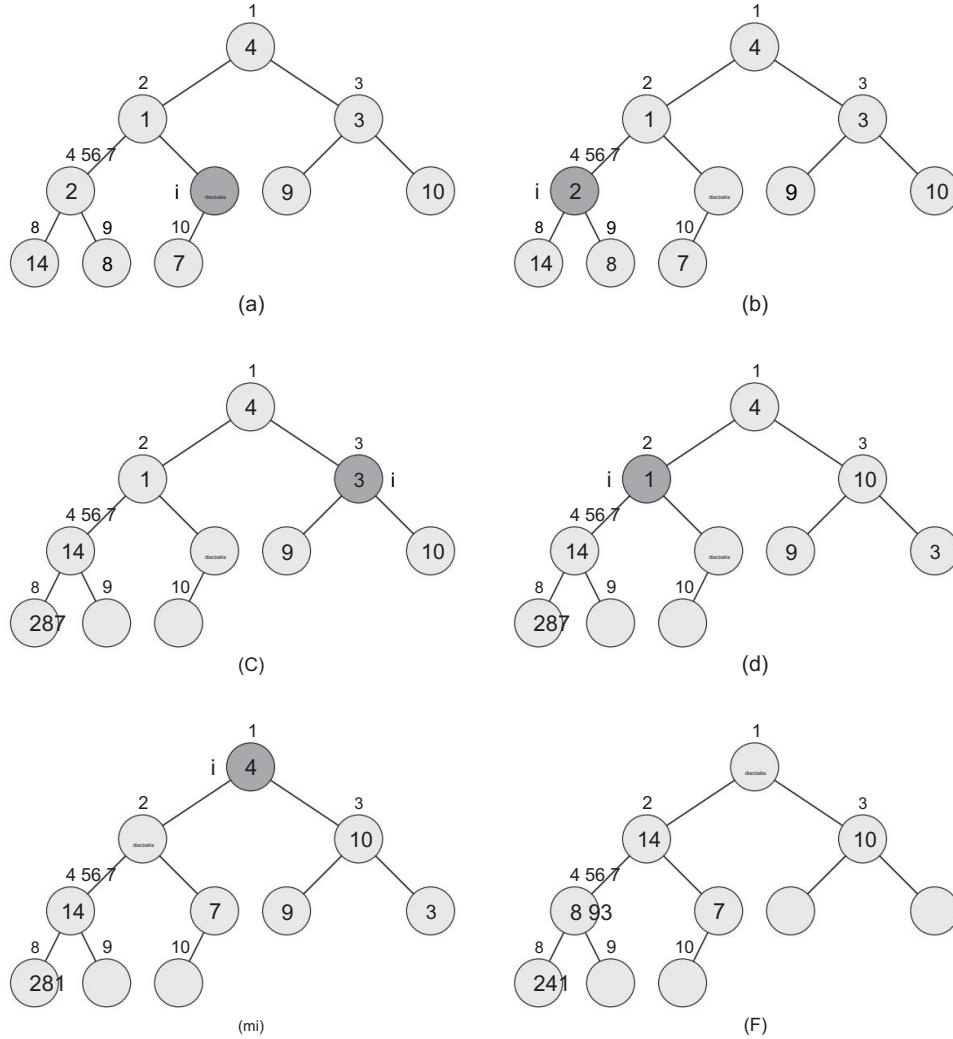


Figura 6.3 La operación de BUILD-MAX-HEAP, que muestra la estructura de datos antes de la llamada a MAX-HEAPIFY en la línea 3 de BUILD-MAX-HEAP. (a) Una matriz de entrada A de 10 elementos y el árbol binario que representa. La figura muestra que el índice de bucle  $i$  se refiere al nodo 5 antes de la llamada MAX-HEAPIFY.A;  $i$ . (b) La estructura de datos resultante. El índice de ciclo  $i$  para la próxima iteración se refiere al nodo 4. (c)–(e) Iteraciones posteriores del ciclo for en BUILD-MAX-HEAP. Observe que cada vez que se llama a MAX-HEAPIFY en un nodo, los dos subáboles de ese nodo son montones máximos. (f) El montón máximo después de que termine BUILD-MAX-HEAP .

$$\text{X}_{\frac{h}{hD0}} \xrightarrow{\text{2 horas}} \text{2hC1 m Oh/ HACER n} \text{X}_{\frac{h}{hD0}} \xrightarrow{\text{2h ! :}} \text{h}$$

Evaluamos la última suma sustituyendo  $x = D = 2$  en la fórmula (A.8), dando

$$\begin{aligned} \text{X}_{\frac{h}{hD0}} & \xrightarrow{\text{2 horas}} \text{D} = \frac{1=2}{.1 1=2/2} \\ & \text{D} = 2 \end{aligned}$$

Por lo tanto, podemos limitar el tiempo de ejecución de BUILD-MAX-HEAP como

$$\begin{aligned} \text{En } \text{X}_{\frac{h}{hD0}} & \xrightarrow{\text{2 horas}} \text{h} \\ & \text{h} \\ & \text{D Encendido/ :} \end{aligned}$$

Por lo tanto, podemos construir un montón máximo a partir de una matriz desordenada en tiempo lineal.

Podemos construir un montón mínimo mediante el procedimiento BUILD-MIN-HEAP, que es lo mismo que BUILD-MAX-HEAP pero con la llamada a MAX-HEAPIFY en la línea 3 reemplazada por una llamada a MIN-HEAPIFY (vea el Ejercicio 6.2-2). BUILD-MIN-HEAP produce un montón mínimo a partir de una matriz lineal desordenada en tiempo lineal.

### Ejercicios

#### 6.3-1

Usando la Figura 6.3 como modelo, ilustre la operación de BUILD-MAX-HEAP en el arreglo  $A[5: 3, 17, 10, 84, 19, 6, 22, 9]$ .

#### 6.3-2

¿Por qué queremos que el índice de ciclo  $i$  en la línea 2 de BUILD-MAX-HEAP disminuya de  $bA:\text{longitud}=2c$  a 1 en lugar de aumentar de 1 a  $bA:\text{longitud}=2c$ ?

#### 6.3-3

Muestre que hay como máximo  $n=2hC1$  nodos de altura  $h$  en cualquier montón de  $n$  elementos.

## 6.4 El algoritmo heapsort

El algoritmo heapsort comienza usando BUILD-MAX-HEAP para construir un montón máximo en la matriz de entrada  $A[1 : : n]$ , donde  $n = A:\text{longitud}$ . Dado que el elemento máximo de la matriz se almacena en la raíz  $A[1]$ , podemos colocarlo en su posición final correcta

intercambiándolo con  $A[i]$ . Si ahora descartamos el nodo  $n$  del montón, y podemos hacerlo simplemente reduciendo  $A[\text{heap-size}]$ , observamos que los hijos de la raíz siguen siendo max-heaps, pero el nuevo elemento raíz podría violar la propiedad max-heap. Sin embargo, todo lo que tenemos que hacer para restaurar la propiedad max-heap es llamar a  $\text{MAX-HEAPIFY}(A, 1)$ , lo que deja un montón máximo en  $A[1 : n - 1]$ . El algoritmo heapsort luego repite este proceso para el montón máximo de tamaño  $n - 1$  hasta un montón de tamaño 2. (Vea el Ejercicio 6.4-2 para un bucle preciso invariante.)

#### HEAPSORT.A/ 1

```
BUILD-MAX-HEAP.A/ 2 for i D
A:longitud hasta 2 3 intercambiar A[1]
con A[4 : i] A:tamaño de pila D
A:tamaño de pila 1 5 MAX-HEAPIFY(A, 1)
```

La Figura 6.4 muestra un ejemplo de la operación de HEAPSORT después de que la línea 1 haya construido el montón inicial máximo. La figura muestra el montón máximo antes de la primera iteración del bucle for de las líneas 2 a 5 y después de cada iteración.

El procedimiento HEAPSORT toma un tiempo  $O(n \lg n)$ , ya que la llamada a BUILD-MAX HEAP toma un tiempo  $O(n)$  y cada una de las  $n - 1$  llamadas a MAX-HEAPIFY toma un tiempo  $O(\lg n)$ .

### Ejercicios

#### 6.4-1

Usando la figura 6.4 como modelo, ilustre la operación de HEAPSORT en el arreglo  $A[5 : 13] = [25, 7, 17, 20, 8, 4]$ .

#### 6.4-2

Argumente la corrección de HEAPSORT usando el siguiente bucle invariante:

Al comienzo de cada iteración del ciclo for de las líneas 2–5, el subarreglo  $A[1 : i]$  es un montón máximo que contiene los  $i$  elementos más pequeños de  $A[1 : n]$ , y el subarreglo  $A[i + 1 : n]$  contiene los  $n - i$  elementos más grandes de  $A[1 : n]$ ,

#### 6.4-3

¿Cuál es el tiempo de ejecución de HEAPSORT en un arreglo  $A$  de longitud  $n$  que ya está clasificado en orden creciente? ¿Qué pasa con el orden decreciente?

#### 6.4-4

Muestre que el peor tiempo de ejecución de HEAPSORT es  $O(n \lg n)$ .

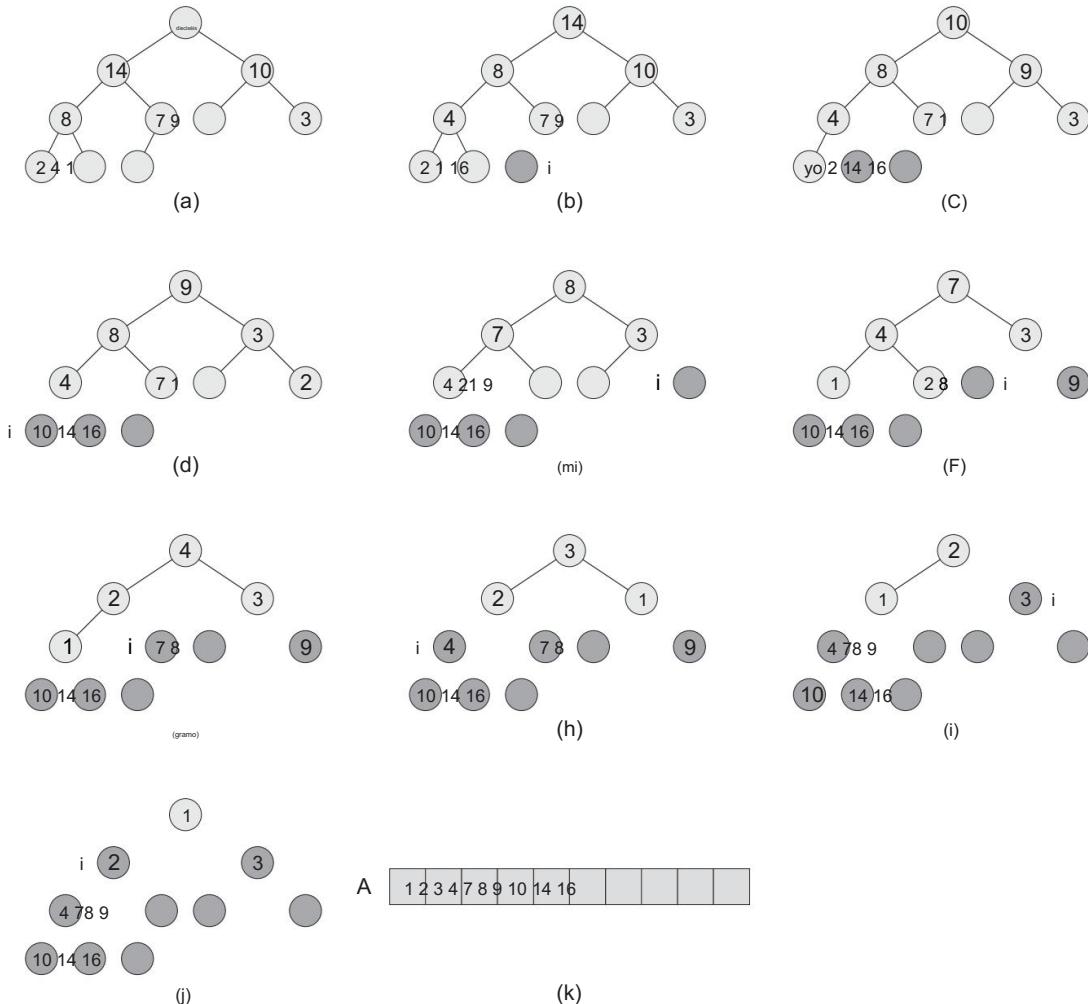


Figura 6.4 El funcionamiento de HEAPSORT. (a) La estructura de datos max-heap justo después de que BUILD-MAX HEAP la haya construido en la línea 1. (b)–(j) Max-heap justo después de cada llamada de MAX-HEAPIFY en la línea 5, que muestra el valor de i En ese tiempo. Solo los nodos ligeramente sombreados permanecen en el montón. (k) La matriz ordenada resultante A.

## 6.4-5 ?

Muestre que cuando todos los elementos son distintos, el mejor tiempo de ejecución de HEAPSORT es  $.n \lg n/$ .

---

## 6.5 Colas de prioridad

Heapsort es un algoritmo excelente, pero una buena implementación de quicksort, presentada en el Capítulo 7, generalmente lo supera en la práctica. No obstante, la estructura de datos del montón en sí misma tiene muchos usos. En esta sección, presentamos una de las aplicaciones más populares de un montón: como cola de prioridad eficiente. Al igual que con los montones, las colas de prioridad vienen en dos formas: colas de prioridad máxima y colas de prioridad mínima. Nos centraremos aquí en cómo implementar colas de prioridad máxima, que a su vez se basan en montones máximos; El ejercicio 6.5-3 le pide que escriba los procedimientos para las colas de prioridad mínima.

Una cola de prioridad es una estructura de datos para mantener un conjunto S de elementos, cada uno con un valor asociado llamado clave . Una cola de máxima prioridad admite las siguientes operaciones:

INSERTAR.S; x/ inserta el elemento x en el conjunto S, que es equivalente al operación SDS [fxg.

MAXIMUM.S / devuelve el elemento de S con la clave más grande.

EXTRACT-MAX.S / elimina y devuelve el elemento de S con la clave más grande.

AUMENTAR-TECLA.S; X; k/ aumenta el valor de la clave del elemento x al nuevo valor k, que se supone que es al menos tan grande como el valor clave actual de x.

Entre sus otras aplicaciones, podemos usar colas de prioridad máxima para programar trabajos en una computadora compartida. La cola de prioridad máxima realiza un seguimiento de los trabajos que se realizarán y sus prioridades relativas. Cuando un trabajo finaliza o se interrumpe, el planificador selecciona el trabajo de mayor prioridad entre los pendientes llamando a EXTRACT-MAX. El planificador puede agregar un nuevo trabajo a la cola en cualquier momento llamando a INSERTAR.

Alternativamente, una cola de prioridad mínima admite las operaciones INSERT, MINIMUM, EXTRACT-MIN y DECREASE-KEY. Se puede usar una cola de prioridad mínima en un simulador controlado por eventos. Los elementos en la cola son eventos a simular, cada uno con un tiempo de ocurrencia asociado que sirve como su clave. Los eventos deben simularse en orden de tiempo de ocurrencia, porque la simulación de un evento puede causar que otros eventos sean simulados en el futuro. El programa de simulación llama a EXTRACT-MIN en cada paso para elegir el próximo evento a simular. A medida que se producen nuevos eventos, el simulador los inserta en la cola de prioridad mínima llamando a INSERT.

Veremos otros usos para las colas de prioridad mínima, destacando la operación DECREASE-KEY , en los capítulos 23 y 24.

No es sorprendente que podamos usar un montón para implementar una cola de prioridad. En una aplicación determinada, como la programación de trabajos o la simulación basada en eventos, los elementos de una cola de prioridad corresponden a los objetos de la aplicación. A menudo necesitamos determinar qué objeto de aplicación corresponde a un elemento de cola de prioridad dado, y viceversa. Cuando usamos un montón para implementar una cola de prioridad, por lo tanto, a menudo necesitamos almacenar un identificador para el objeto de la aplicación correspondiente en cada elemento del montón. La composición exacta del identificador (como un puntero o un número entero) depende de la aplicación. De manera similar, necesitamos almacenar un identificador para el elemento de montón correspondiente en cada objeto de aplicación. Aquí, el identificador normalmente sería un índice de matriz. Debido a que los elementos del montón cambian de ubicación dentro de la matriz durante las operaciones del montón, una implementación real, al reubicar un elemento del montón, también tendría que actualizar el índice de la matriz en el objeto de la aplicación correspondiente. Debido a que los detalles del acceso a los objetos de la aplicación dependen en gran medida de la aplicación y su implementación, no los analizaremos aquí, aparte de señalar que, en la práctica, estos identificadores deben mantenerse correctamente.

Ahora discutimos cómo implementar las operaciones de una cola de máxima prioridad. El procedimiento HEAP-MAXIMUM implementa la operación MAXIMUM en ,1/ tiempo.

HEAP-MAXIMUM.A/ 1

retorno ACE1

El procedimiento HEAP-EXTRACT-MAX implementa la opera EXTRACT-MAX  
ción Es similar al cuerpo del bucle for (líneas 3 a 5) del procedimiento HEAPSORT .

HEAP-EXTRACT-MAX.A/ 1 si

A:heap-size < 1 2 error  
"heap underflow" 3 max D ACE1 4 ACE1  
D ACEA:heap-size  
5 A:heap-size D A:heap-size 1 6  
MAX -HEAPIFY.A; 1/ 7 devolución máx.

El tiempo de ejecución de HEAP-EXTRACT-MAX es O.lg n/, ya que realiza solo una cantidad constante de trabajo además del tiempo O.lg n/ de MAX-HEAPIFY.

El procedimiento HEAP-INCREASE-KEY implementa la operación INCREASE-KEY . Un índice i en la matriz identifica el elemento de cola de prioridad cuya clave deseamos aumentar. El procedimiento primero actualiza la clave del elemento ACEi a su nuevo valor. Debido a que aumentar la clave de ACEi podría violar la propiedad max-heap,

Entonces, el procedimiento, de una manera que recuerda al ciclo de inserción (líneas 5 a 7) de INSERCIÓN-CLASIFICACIÓN de la Sección 2.1, recorre un camino simple desde este nodo hacia la raíz para encontrar un lugar adecuado para la clave recién aumentada. A medida que HEAP-INCREASE KEY atraviesa esta ruta, compara repetidamente un elemento con su parente, intercambia sus claves y continúa si la clave del elemento es más grande, y termina si la clave del elemento es más pequeña, ya que ahora se mantiene la propiedad max-heap. (Consulte el ejercicio 6.5-5 para obtener una invariante de ciclo precisa).

```
MONTÓN-AUMENTO-CLAVE.A; i; clave/ 1
si la clave < AŒi 2
error "la nueva clave es más pequeña que la clave actual"
3 AŒi D tecla 4
mientras i>1 y ACEPARENT.i / < AŒi 5 intercambia
AŒi con ACEPARENT.i / 6 i D PARENT.i /
```

La figura 6.5 muestra un ejemplo de una operación HEAP-INCREASE-KEY . El tiempo de ejecución de HEAP-INCREASE-KEY en un montón de n elementos es O.lg n/, ya que la ruta trazada desde el nodo actualizado en la línea 3 hasta la raíz tiene una longitud O.lg n/.

El procedimiento MAX-HEAP-INSERT implementa la operación INSERT . Toma como entrada la clave del nuevo elemento que se insertará en max-heap A. El procedimiento primero expande max-heap agregando al árbol una nueva hoja cuya clave es 1.

Luego llama a HEAP-INCREASE-KEY para establecer la clave de este nuevo nodo en su valor correcto y mantener la propiedad max-heap.

```
MAX-HEAP-INSERT.A; key/ 1 A:tamaño de
almacenamiento dinámico D A:tamaño de almacenamiento
dinámico C 1 2 ACEA:tamaño de
almacenamiento dinámico D 1 3 HEAP-INCREASE-KEY.A; A: tamaño del montón; llave/
```

El tiempo de ejecución de MAX-HEAP-INSERT en un montón de n elementos es O.lg n/.

En resumen, un montón puede admitir cualquier operación de cola de prioridad en un conjunto de tamaño n en O.lg n/ time.

### Ejercicios

#### 6.5-1

Ilustre el funcionamiento de HEAP-EXTRACT-MAX en el montón AD h15; 13; 9; 5; 12; 8; 7; 4; 0; 6; 2; 1i.

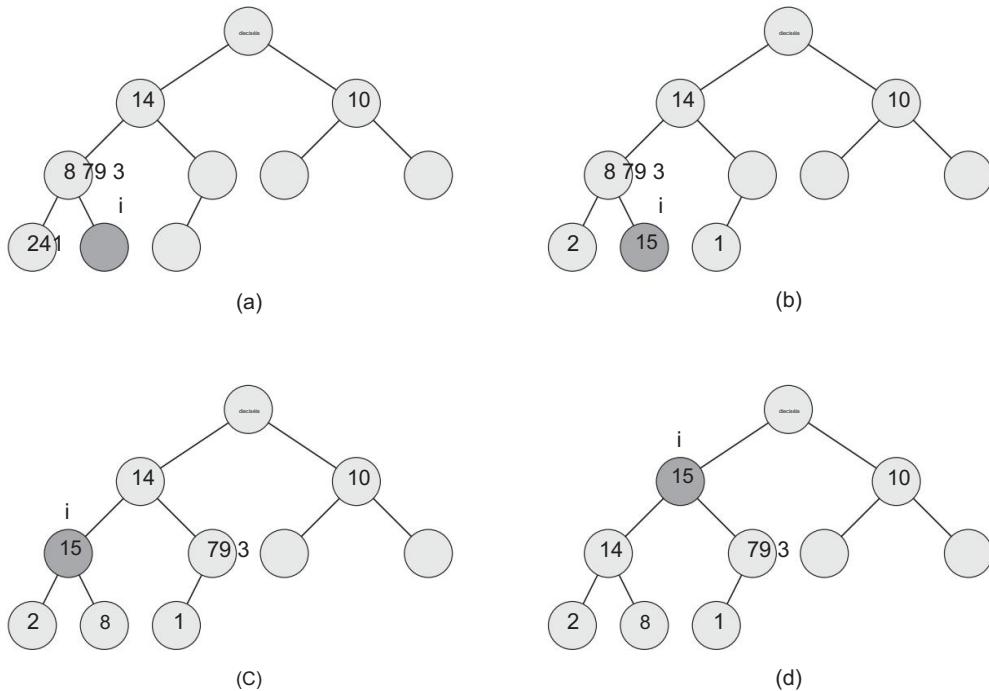


Figura 6.5 El funcionamiento de HEAP-INCREASE-KEY. (a) El max-heap de la Figura 6.4(a) con un nodo cuyo índice está fuertemente sombreado. (b) Este nodo tiene su clave aumentada a 15. (c) Despues de una iteración del bucle while de las líneas 4–6, el nodo y su padre han intercambiado claves, y el índice  $i$  sube al padre. (d) El montón máximo despues de una iteración más del ciclo while . En este punto,  $A[\text{PARENT}.i] < A[i]$ . La propiedad max-heap ahora se mantiene y el procedimiento finaliza.

6.5-2

Ilustrar el funcionamiento de MAX-HEAP-INSERT.A; 10/ en el montón AD h15; 13; 9; 5; 12; 8; 7; 4; 0; 6; 2; 1.

6.5-3

Escriba pseudocódigo para los procedimientos HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY y MIN-HEAP-INSERT que implementan una cola de prioridad mínima con un montón mínimo.

65-4

¿Por qué nos molestamos en establecer la clave del nodo insertado en 1 en la línea 2 de MAX-HEAP-INSERT cuando lo siguiente que hacemos es aumentar su clave al valor deseado?

## 6.5-5

Argumente la corrección de HEAP-INCREASE-KEY usando el siguiente bucle invariante:

Al comienzo de cada iteración del ciclo while de las líneas 4–6, el subarreglo A[1 : : i] satisface la propiedad max-heap, excepto que puede haber una violación: A[i] puede ser mayor que A[PARENT(i)] .

Puede suponer que el subarreglo A[1 : : i] satisface la propiedad max-heap en el momento en que se llama HEAP-INCREASE-KEY .

## 6.5-6

Cada operación de intercambio en la línea 5 de HEAP-INCREASE-KEY normalmente requiere tres asignaciones. Muestre cómo usar la idea del ciclo interno de CLASIFICACIÓN POR INSERCIÓN para reducir las tres asignaciones a una sola asignación.

## 6.5-7

Muestre cómo implementar una cola de primero en entrar, primero en salir con una cola de prioridad. Muestre cómo implementar una pila con una cola de prioridad. (Las colas y las pilas se definen en la Sección 10.1.)

## 6.5-8

La operación HEAP-DELETE(A; i) elimina el elemento en el nodo i del montón A. Proporcione una implementación de HEAP-DELETE que se ejecuta en  $O(\lg n)$  time para un montón máximo de n elementos.

## 6.5-9

Proporcione un algoritmo  $O(n \lg k)$ -time para fusionar k listas ordenadas en una lista ordenada, donde n es el número total de elementos en todas las listas de entrada. (Sugerencia: use un montón mínimo para la fusión de k-way).

---

## Problemas

### 6-1 Construyendo un montón usando inserción

Podemos construir un montón llamando repetidamente a MAX-HEAP-INSERT para insertar los elementos en el montón. Considere la siguiente variación del procedimiento BUILD-MAX-HEAP :

BUILD-MAX-HEAP0 .A/ 1 A:

tamaño de almacenamiento  
dinámico D 1 2 para i D 2 a A:  
longitud 3 MAX-HEAP-INSERT.A; AŒi/

a. ¿Los procedimientos BUILD-MAX-HEAP y BUILD-MAX-HEAP0 siempre crean el mismo montón cuando se ejecutan en la misma matriz de entrada? Demuestre que lo hacen, o proporcione un contrajeemplo.

b. Muestre que en el peor de los casos, BUILD-MAX-HEAP0 requiere  $n \lg n$  tiempo para construir un montón de  $n$  elementos.

#### 6-2 Análisis de montones d-ario Un

montón d-ario es como un montón binario, pero (con una posible excepción) los nodos que no son hojas tienen d hijos en lugar de 2 hijos.

a. ¿Cómo representarías un montón d-ario en una matriz?

b. ¿Cuál es la altura de un montón d-ario de  $n$  elementos en términos de  $n$  y  $d$ ?

C. Proporcione una implementación eficiente de EXTRACT-MAX en un d-ary max-heap. Un analice su tiempo de ejecución en términos de  $d$  y  $n$ .

d. Proporcione una implementación eficiente de INSERT en un d-ary max-heap. Analice su tiempo de ejecución en términos de  $d$  y  $n$ .

mi. Dar una implementación eficiente de INCREASE-KEY.A; i; k/, que marca un error si  $k < AŒi$ , pero de lo contrario establece  $AŒi$  D  $k$  y luego actualiza la estructura de almacenamiento dinámico d-ary max de manera adecuada. Analice su tiempo de ejecución en términos de  $d$  y  $n$ .

#### 6-3 Cuadros Young Un

cuadro mn Young es una matriz mn tal que las entradas de cada fila están ordenadas de izquierda a derecha y las entradas de cada columna están ordenadas de arriba abajo. Algunas de las entradas de un cuadro de Young pueden ser 1, que tratamos como elementos inexistentes. Por lo tanto, se puede usar un cuadro de Young para contener r mn números finitos.

a. Dibuje un cuadro de 44 Young que contenga los elementos f9; dieciséis; 3; 2; 4; 8; 5; 14; 12 g.

b. Argumente que un cuadro Y de mn Young está vacío si  $Y \subset 1; 1 D 1$ . Argumentar que S está lleno (contiene mn elementos) si  $Y \subset m; n < 1$ .

C. Proporcione un algoritmo para implementar EXTRACT-MIN en un cuadro de mn Young no vacío que se ejecuta en tiempo  $O(m C n)$ . Su algoritmo debe usar una subrutina recursiva que resuelva un problema mn resolviendo recursivamente un subproblema  $m 1/n \leq m \leq n/1$ . (Sugerencia: piense en MAX HEAPIFY.) Defina  $T(p)$ , donde  $p \leq D \leq C \leq n$ , como el tiempo máximo de ejecución de EXTRACT-MIN en cualquier cuadro de mn Young. Dé y resuelva una recurrencia para  $T(p)$  que produzca el límite de tiempo  $O(m C n)$ .

d. Muestre cómo insertar un nuevo elemento en un cuadro de mn Young no completo en tiempo  $O(m C n)$ .

mi. Sin usar otro método de clasificación como subrutina, muestre cómo usar un nn Young tableau para ordenar  $n^2$  números en  $O(n^3)$  tiempo.

F. Proporcione un algoritmo  $O(m C n)$ -time para determinar si un número dado se almacena en un cuadro dado de mn Young.

### Notas del capítulo

El algoritmo heapsort fue inventado por Williams [357], quien también describió cómo implementar una cola de prioridad con un montón. El procedimiento BUILD-MAX-HEAP fue sugerido por Floyd [106].

Usamos montones mínimos para implementar colas de prioridad mínima en los capítulos 16, 23 y 24. También damos una implementación con límites de tiempo mejorados para ciertas operaciones en el Capítulo 19 y, suponiendo que las claves se extraen de un conjunto acotado de enteros no negativos, el Capítulo 20.

Si los datos son enteros de b-bit y la memoria de la computadora consta de palabras de b-bit direccionables, Fredman y Willard [115] mostraron cómo implementar MINIMUM en  $O(1)$  time e INSERT y EXTRACT-MIN en  $O(\lg n)$  time. Thorup [337] ha mejorado el tiempo  $O(\lg n)$  enlazado a  $O(\lg \lg n)$ . Este límite utiliza una cantidad de espacio ilimitado en  $n$ , pero se puede implementar en un espacio lineal mediante el uso de hashing aleatorio.

Un caso especial importante de colas de prioridad ocurre cuando la secuencia de operaciones EXTRACT-MIN es monótona, es decir, los valores devueltos por operaciones EXTRACT-MIN sucesivas aumentan monótonamente con el tiempo. Este caso surge en varias aplicaciones importantes, como el algoritmo de caminos más cortos de fuente única de Dijkstra, que analizamos en el capítulo 24, y en la simulación de eventos discretos. Para el algoritmo de Dijkstra es particularmente importante que la operación DECREASE-KEY se implemente de manera eficiente. Para el caso monótono, si los datos son números enteros en el rango  $1; 2; \dots; C$ , Ahuja, Mehnhorn, Orlin y Tarjan [8] describen

cómo implementar EXTRACT-MIN e INSERT en  $O.\lg C / \text{tiempo amortizado}$  (consulte el Capítulo 17 para obtener más información sobre el análisis amortizado) y DECREASE-KEY en  $O.1/\sqrt{\lg C}$ , usando una estructura de datos llamada montón de base. El límite  $O.\lg C / \sqrt{\lg C}$  se puede mejorar a  $O.\lg C / \sqrt{\lg \lg C}$  usando montones de Fibonacci (consulte el Capítulo 19) junto con montones de base. Cherkassky, Goldberg y Silverstein [65] mejoraron aún más el límite a  $O.\lg 1=3C \lg C / \text{tiempo esperado}$  al combinar la estructura de depósito multinivel de Denardo y Fox [85] con el montón de Thorup mencionado anteriormente. Raman [291] mejoró aún más estos resultados para obtener un límite de  $O.\min(\lg 1=4C, \lg 1=3C) n^{1/2}$ , para cualquier fijo  $>0$ .

---

# 7

## Ordenación rápida

El algoritmo de clasificación rápida tiene un tiempo de ejecución en el peor de los casos de  $n^2$  en una matriz de entrada de  $n$  números. A pesar de este tiempo de ejecución lento en el peor de los casos, quicksort es a menudo la mejor opción práctica para clasificar porque es notablemente eficiente en promedio: su tiempo de ejecución esperado es  $n \lg n$ , y los factores constantes ocultos en  $n \lg n$  La notación  $n$  es bastante pequeña. También tiene la ventaja de ordenar en el lugar (consulte la página 17) y funciona bien incluso en entornos de memoria virtual.

La Sección 7.1 describe el algoritmo y una subrutina importante utilizada por la ordenación rápida para la partición. Debido a que el comportamiento de ordenación rápida es complejo, comenzamos con una discusión intuitiva de su desempeño en la Sección 7.2 y posponemos su análisis preciso hasta el final del capítulo. La Sección 7.3 presenta una versión de ordenación rápida que usa muestreo aleatorio. Este algoritmo tiene un buen tiempo de ejecución esperado y ninguna entrada en particular provoca su comportamiento en el peor de los casos. La Sección 7.4 analiza el algoritmo aleatorio, mostrando que se ejecuta en el tiempo  $n^2$  en el peor de los casos y, suponiendo elementos distintos, en el tiempo esperado  $O(n \lg n)$ .

---

### 7.1 Descripción de ordenación rápida

Quicksort, al igual que merge sort, aplica el paradigma divide y vencerás presentado en la Sección 2.3.1. Aquí está el proceso de dividir y vencer de tres pasos para clasificar un subarreglo  $A[p:r]$ :

**Dividir:** dividir (reorganizar) el arreglo  $A[p:r]$  en dos (posiblemente vacíos) rayos de subarreglos  $A[p:q]$  y  $A[q:r]$  de tal manera que cada elemento de  $A[p:q]$  sea menor o igual que  $A[q]$ , que es, a su vez, menor o igual que cada elemento de  $A[q:r]$ . Calcule el índice  $q$  como parte de este procedimiento de partición.

**Conquer:** ordenar los dos subarreglos  $A[p:q]$  y  $A[q:r]$  mediante llamadas recursivas para clasificar rápidamente.

Combinar: debido a que los subarreglos ya están ordenados, no se necesita trabajo para combinarlos:  
el arreglo completo  $A[p:r]$  ahora está ordenado.

El siguiente procedimiento implementa ordenación rápida:

```
QUICKSORT.A; pag; r/ 1
si p<r q D
    PARTICION.A; pag; r/ 2 3
    QUICKSORT.A; pag; q 1/ 4 QUICKSORT.A;
    q C 1; r/
```

Para ordenar una matriz A completa, la llamada inicial es  $\text{QUICKSORT.A}(A, 1, \text{length}(A))$ .

#### Partición de la matriz

La clave del algoritmo es el procedimiento de **PARTICIÓN**, que reorganiza el subarreglo  $A[p:r]$  en su lugar.

```
PARTICIÓN.A; pag; r/ 1
x D A[1:r]
p 1 3 para j D p a
r 1 si A[j] < x 4 i D i C 1
    intercambia
    A[i] con A[j]
    6 7 intercambia
    A[i] C 1 con A[r] regresa i
C 1
```

La figura 7.1 muestra cómo funciona **PARTICIÓN** en una matriz de 8 elementos. **PARTITION** siempre selecciona un elemento  $x = A[1:r]$  como elemento pivote alrededor del cual dividir el subarreglo  $A[p:r]$ . A medida que se ejecuta el procedimiento, divide la matriz en cuatro regiones (posiblemente vacías). Al comienzo de cada iteración del ciclo for en las líneas 3 a 6, las regiones satisfacen ciertas propiedades, que se muestran en la figura 7.2. Enunciamos estas propiedades como un bucle invariante:

Al comienzo de cada iteración del bucle de las líneas 3 a 6, para cualquier índice de matriz  $k$ ,

1. Si  $p \leq k \leq r$ , entonces  $A[k] = x$ .
2. Si  $i \leq k \leq j - 1$ , entonces  $A[k] > x$ .
3. Si  $k < i$  o  $k > j$ , entonces  $A[k] \leq x$ .

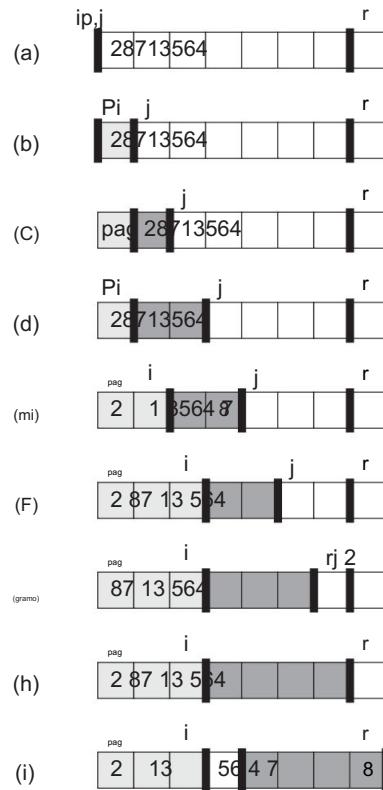


Figura 7.1 La operación de PARTICIÓN en un arreglo de muestra. La entrada de matriz AOE se convierte en el elemento pivote x. Los elementos de matriz ligeramente sombreados están todos en la primera partición con valores no mayores que x.

Los elementos muy sombreados están en la segunda partición con valores mayores que x. Los elementos sin sombrear aún no se han colocado en una de las dos primeras particiones, y el último elemento blanco es el pivote x. (a) La matriz inicial y la configuración de las variables. Ninguno de los elementos se ha colocado en ninguno de los dos primeros tabiques. (b) El valor 2 se “permute consigo mismo” y se coloca en la partición de valores más pequeños. (c)–(d) Los valores 8 y 7 se suman a la partición de valores más grandes. (e) Los valores 1 y 8 se intercambian y la partición más pequeña crece. (f) Los valores 3 y 7 se intercambian y la partición más pequeña crece. (g)–(h) La partición más grande crece para incluir 5 y 6, y el bucle termina. (i) En las líneas 7 y 8, el elemento de pivote se intercambia para que quede entre las dos particiones.

Los índices entre  $j$  y  $r$  no están cubiertos por ninguno de los tres casos, y los valores en estas entradas no tienen una relación particular con el pivote x.

Necesitamos mostrar que esta invariantes del ciclo es verdadera antes de la primera iteración, que cada iteración del ciclo mantiene la invariantes y que la invariantes proporciona una propiedad útil para mostrar la corrección cuando termina el ciclo.

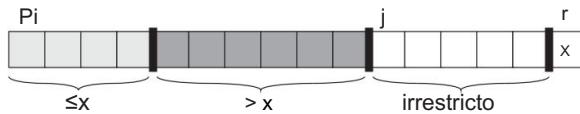


Figura 7.2 Las cuatro regiones mantenidas por el procedimiento PARTICIÓN en un subarreglo  $A[p:r]$ . Los valores de  $A[i:j]$  son todos menores o iguales que  $x$ , los valores de  $A[i+1:j]$  son todos mayores que  $x$ , y  $A[r]$  es  $x$ . El subarreglo  $A[i:r]$  puede tomar cualquier valor.

**Inicialización:** antes de la primera iteración del ciclo,  $i = p$  y  $j = p$ . Debido a que no hay valores entre  $p$  e  $i$  y no hay valores entre  $i+1$  y  $j$ , las dos primeras condiciones del ciclo invariante se satisfacen trivialmente. La asignación en la línea 1 satisface la tercera condición.

**Mantenimiento:** Como muestra la Figura 7.3, consideraremos dos casos, dependiendo del resultado de la prueba en la línea 4. La Figura 7.3(a) muestra lo que sucede cuando  $A[j] > x$ ; la única acción en el ciclo es incrementar  $j$ . Después de que se incrementa  $j$ , la condición 2 se cumple para  $A[i:j]$  y todas las demás entradas permanecen sin cambios. La Figura 7.3(b) muestra lo que sucede cuando  $A[j] \leq x$ ; el bucle incrementa  $i$ , intercambia  $A[i]$  y  $A[j]$  y luego incrementa  $j$ . Debido al intercambio, ahora tenemos que  $A[i] \leq x$ , y se cumple la condición 1. De manera similar, también tenemos que  $A[j] > x$ , ya que el elemento que se intercambió en  $A[j]$  es, por el bucle invariante, mayor que  $x$ .

**Terminación:** En la terminación,  $j = r$ . Por lo tanto, cada entrada en el arreglo está en uno de los tres conjuntos descritos por el invariante, y hemos dividido los valores del arreglo en tres conjuntos: los menores o iguales a  $x$ , los mayores que  $x$  y un conjunto único que contiene  $x$ .

Las últimas dos líneas de PARTITION terminan intercambiando el elemento pivote con el elemento más a la izquierda mayor que  $x$ , moviendo así el pivote a su lugar correcto en la matriz particionada y luego devolviendo el nuevo índice del pivote. La salida de PARTICIÓN ahora cumple con las especificaciones dadas para el paso de división. De hecho, satisface una condición un poco más fuerte: después de la línea 2 de QUICKSORT,  $A[q]$  es estrictamente menor que cada elemento de  $A[q:C-1]$ .

El tiempo de ejecución de PARTICIÓN en el subarreglo  $A[p:r]$  es  $\sim n$ , donde  $n = r - p + 1$  (vea el Ejercicio 7.1-3).

## Ejercicios

### 7.1-1

Usando la Figura 7.1 como modelo, ilustre la operación de PARTICIÓN en el arreglo  $A[13:19; 9:5; 12:8; 7:4; 21:2; 6:11]$ .

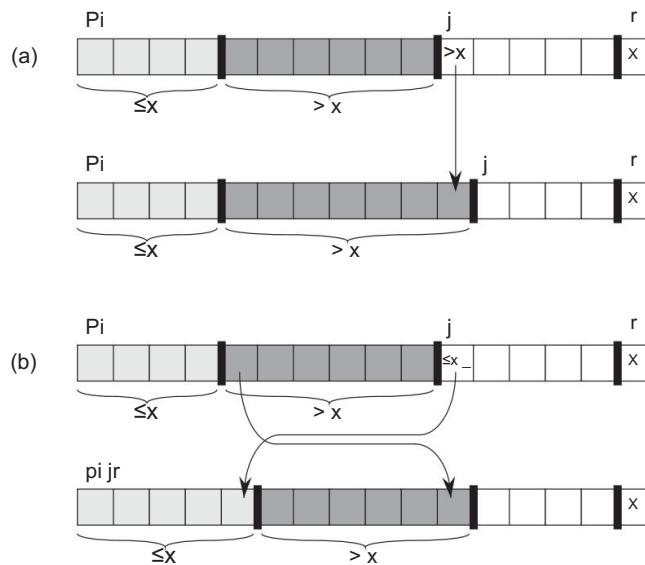


Figura 7.3 Los dos casos para una iteración del procedimiento PARTICIÓN. (a) Si  $A[i] > x$ , la única acción es incrementar  $j$ , lo que mantiene el bucle invariante. (b) Si  $A[i] = x$ , el índice  $i$  se incrementa,  $A[i]$  y  $A[j]$  se intercambian, y luego se incrementa  $j$ . De nuevo, se mantiene el ciclo invariante.

#### 7.1-2

¿Qué valor de  $q$  devuelve PARTICIÓN cuando todos los elementos del arreglo  $A[0:r]$  tienen el mismo valor? Modifique PARTICIÓN para que  $q \leftarrow r$  cuando todos los elementos en el arreglo  $A[0:r]$  tengan el mismo valor.

#### 7.1-3

Dé un breve argumento de que el tiempo de ejecución de PARTITION en un subarreglo de tamaño  $n$  es  $\Theta(n)$ .

#### 7.1-4

¿Cómo modificaría QUICKSORT para clasificar en orden no creciente?

## 7.2 Rendimiento de clasificación rápida

El tiempo de ejecución de quicksort depende de si la partición está equilibrada o no, lo que a su vez depende de qué elementos se utilizan para la partición.

Si la partición está equilibrada, el algoritmo se ejecuta asintóticamente tan rápido como merge

clasificar. Sin embargo, si la partición está desequilibrada, puede ejecutarse asintóticamente tan lentamente como la ordenación por inserción. En esta sección, investigaremos de manera informal cómo funciona Quicksort bajo los supuestos de partición balanceada versus no balanceada.

#### Partición en el peor de los casos

El comportamiento en el peor de los casos para la ordenación rápida ocurre cuando la rutina de partición produce un subproblema con  $n-1$  elementos y uno con 0 elementos. (Probamos esta afirmación en la Sección 7.4.1.) Supongamos que esta partición desequilibrada surge en cada llamada recursiva. La partición cuesta  $C_n$  tiempo. Dado que la llamada recursiva en una matriz de tamaño 0 simplemente regresa,  $T(0) = D$ , y la recurrencia para el tiempo de ejecución es

$$T(n) = C_n + CT(n-1)$$

Intuitivamente, si sumamos los costos incurridos en cada nivel de la recursividad, obtenemos una serie aritmética (ecuación (A.2)), que se evalúa como  $\frac{n(n+1)}{2}$ . De hecho, es sencillo usar el método de sustitución para probar que la recurrencia  $T(n) = C_n + CT(n-1)$  tiene la solución  $T(n) = D + \frac{C}{2}(n^2 - 1)$ . (Consulte el ejercicio 7.2-1.)

Por lo tanto, si la partición está desequilibrada al máximo en cada nivel recursivo del algoritmo, el tiempo de ejecución es  $\frac{n^2}{2}$ . Por lo tanto, el tiempo de ejecución en el peor de los casos de ordenación rápida no es mejor que el de ordenación por inserción. Además, el tiempo de ejecución  $\frac{n^2}{2}$  ocurre cuando la matriz de entrada ya está completamente ordenada, una situación común en la que la ordenación por inserción se ejecuta en el tiempo  $O(n)$ .

#### Partición en el mejor de los casos

En la división más uniforme posible, PARTITION produce dos subproblemas, cada uno de tamaño no mayor que  $n/2$ , ya que uno es de tamaño  $b=2c$  y otro de tamaño  $d=n-2c$ . En este caso, quicksort se ejecuta mucho más rápido. La recurrencia para el tiempo de ejecución es entonces

$$T(n) = T(n/2) + O(n)$$

donde toleramos el descuido de ignorar el piso y el techo y de restar 1. Por el caso 2 del teorema maestro (Teorema 4.1), esta recurrencia tiene la solución  $T(n) = O(n \lg n)$ . Equilibrando por igual los dos lados de la partición en cada nivel de la recursividad, obtenemos un algoritmo asintóticamente más rápido.

#### Partición equilibrada

El tiempo de ejecución del caso promedio de ordenación rápida está mucho más cerca del mejor de los casos que del peor de los casos, como lo mostrarán los análisis en la Sección 7.4. La clave para entender -

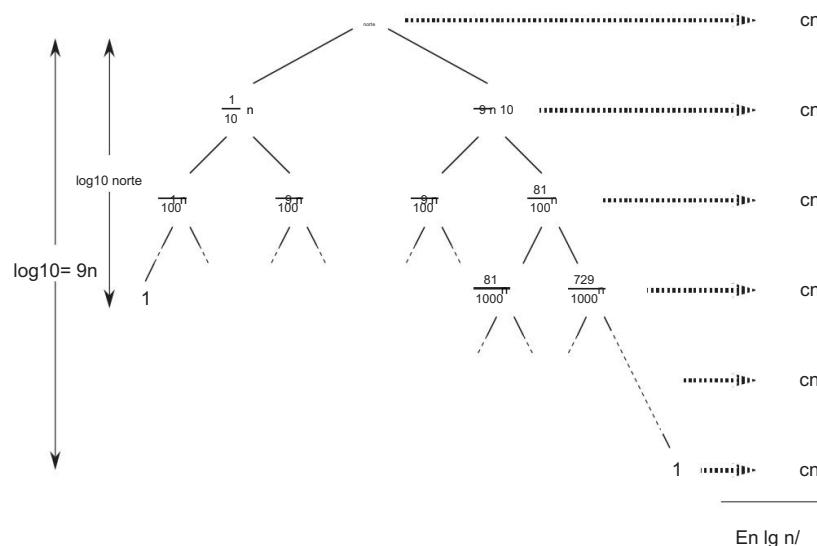


Figura 7.4 Un árbol de recurrencia para QUICKSORT en el que PARTITION siempre produce una división de 9 a 1, lo que produce un tiempo de ejecución de  $O(n \lg n)$ . Los nodos muestran los tamaños de los subproblemas, con costos por nivel a la derecha. Los costos por nivel incluyen la constante  $c$  implícita en el término  $.n/$ .

Explicar por qué es comprender cómo el equilibrio de la partición se refleja en la recurrencia que describe el tiempo de ejecución.

Supongamos, por ejemplo, que el algoritmo de partición siempre produce una división proporcional de 9 a 1, que a primera vista parece bastante desequilibrada. Obtenemos entonces la reaparición

$$T(n) = DT(9n) + CT(n) + cn$$

sobre el tiempo de ejecución de quicksort, donde hemos incluido explícitamente la constante  $c$  escondida en el término  $.n/$ . La Figura 7.4 muestra el árbol de recurrencia para esta recurrencia. Observe que cada nivel del árbol tiene un costo  $c_n$ , hasta que la recurrencia alcanza una condición límite en la profundidad  $\log_{10} n = 9$ ,  $\lg n$ , y luego los niveles tienen un costo como máximo  $c_n$ . La recursividad termina en la profundidad  $\log_{10} n = 9$ ,  $\lg n$ . Por lo tanto, el costo total de la clasificación rápida es  $O(n \lg n)$ . Por lo tanto, con una división proporcional de 9 a 1 en cada nivel de recursividad, que intuitivamente parece bastante desequilibrada, la ordenación rápida se ejecuta en  $O(n \lg n)$  time, asintóticamente igual que si la división estuviera justo en el medio. De hecho, incluso una división de 99 a 1 produce un tiempo de ejecución  $O(n \lg n)$ . De hecho, cualquier división de proporcionalidad constante produce un árbol recursivo de profundidad  $\lg n$ , donde el costo en cada nivel es  $O(n)$ . Por lo tanto, el tiempo de ejecución es  $O(n \lg n)$  siempre que la división tenga una proporcionalidad constante.

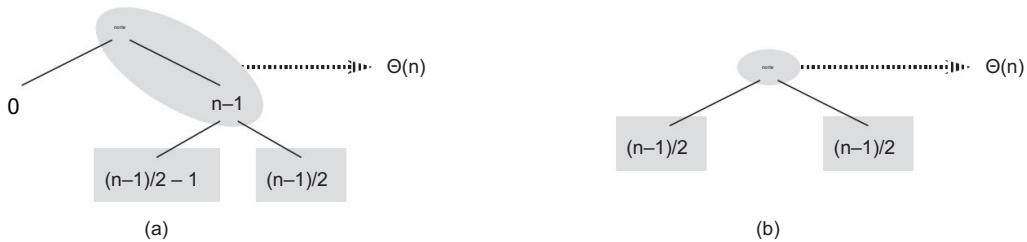


Figura 7.5 (a) Dos niveles de un árbol recursivo para clasificación rápida. La partición en la raíz cuesta  $n$  y produce una división "mala": dos subarreglos de tamaño 0 y  $n$ . La partición del subarreglo de tamaño  $n$  1 cuesta  $n$  1 y produce una división "buena": subarreglos de tamaño  $.n 1/2$  y  $.n 1/2$ . (b) Un solo nivel de un árbol de recurrencia que está muy bien balanceado. En ambas partes, el costo de partición de los subproblemas mostrados con sombreado elíptico es  $,n/$ . Sin embargo, los subproblemas que quedan por resolver en (a), que se muestran con sombreado cuadrado, no son más grandes que los correspondientes subproblemas que quedan por resolver en (b).

#### Intuición para el caso promedio

Para desarrollar una noción clara del comportamiento aleatorio de ordenación rápida, debemos hacer una suposición sobre la frecuencia con la que esperamos encontrar las distintas entradas. El comportamiento de la ordenación rápida depende del orden relativo de los valores en los elementos de la matriz dados como entrada, y no de los valores particulares de la matriz. Como en nuestro análisis probabilístico del problema de contratación en la Sección 5.2, supondremos por ahora que todas las permutaciones de los números de entrada son igualmente probables.

Cuando ejecutamos quicksort en una matriz de entrada aleatoria, es muy poco probable que la partición suceda de la misma manera en todos los niveles, como lo ha supuesto nuestro análisis informal. Esperamos que algunas de las divisiones estén razonablemente bien equilibradas y que otras estén bastante desequilibradas. Por ejemplo, el ejercicio 7.2-6 le pide que demuestre que alrededor del 80 por ciento de las veces PARTITION produce una división que está más balanceada que 9 a 1, y alrededor del 20 por ciento de las veces produce una división que está menos balanceada que 9 a 1 .

En el caso promedio, PARTITION produce una mezcla de divisiones "buenas" y "malas". En un árbol de recurrencia para una ejecución de caso promedio de PARTICIÓN, las divisiones buenas y malas se distribuyen aleatoriamente por todo el árbol. Supongamos, en aras de la intuición, que las divisiones buenas y malas alternan niveles en el árbol, y que las divisiones buenas son divisiones en el mejor de los casos y las divisiones malas son divisiones en el peor de los casos. La Figura 7.5(a) muestra las divisiones en dos niveles consecutivos en el árbol de recurrencia. En la raíz del árbol, el costo de partición es  $n$ , y los subarreglos producidos tienen tamaños  $n$  1 y 0: el peor de los casos. En el siguiente nivel, el subarreglo de tamaño  $n$  1 se divide en el mejor de los casos en subarreglos de tamaño  $.n 1/2$  y  $.n 1/2$ . Supongamos que el costo de la condición de contorno es 1 para el subarreglo de tamaño 0.

La combinación de la mala división seguida de la buena división produce tres subarreglos de tamaños  $0, \frac{n}{2}, 1$  y  $\frac{n}{2}$  con un costo de partición combinado de  $\frac{n}{2}C + \frac{n}{2}D + n$ . Ciertamente, esta situación no es peor que la de la Figura 7.5(b), es decir, un solo nivel de partición que produce dos subarreglos de tamaño  $\frac{n}{2}$ , a un costo de  $n$ . Sin embargo, esta última situación está equilibrada! Intuitivamente, el costo  $n/2$  de la mala división puede absorberse en el costo  $n$  de la buena división, y la división resultante es buena. Por lo tanto, el tiempo de ejecución de quicksort, cuando los niveles alternan entre buenos y malos splits, es como el tiempo de ejecución de los buenos splits solos: todavía es  $O(n \lg n)$ , pero con una constante ligeramente mayor oculta por la notación  $O$ .

Daremos un análisis riguroso del tiempo de ejecución esperado de una versión aleatoria de clasificación rápida en la Sección 7.4.2.

### Ejercicios

#### 7.2-1

Use el método de sustitución para probar que la recurrencia  $T(n) = DT(n/2) + Cn + Dn^2$  tiene la solución  $T(n) = O(n^2)$ , como se afirmó al comienzo de la Sección 7.2.

#### 7.2-2

¿Cuál es el tiempo de ejecución de QUICKSORT cuando todos los elementos del arreglo A tienen el mismo valor?

#### 7.2-3

Muestre que el tiempo de ejecución de QUICKSORT es  $n^2$  cuando el arreglo A contiene elementos distintos y está ordenado en orden decreciente.

#### 7.2-4

Los bancos a menudo registran las transacciones en una cuenta en el orden de los tiempos de la transacciones, pero a muchas personas les gusta recibir sus extractos bancarios con cheques enumerados en orden por número de cheque. La gente suele escribir cheques en orden por número de cheque, y los comerciantes suelen cobrarlos con una rapidez razonable. El problema de convertir el pedido por tiempo de transacción en un pedido por número de cheque es, por lo tanto, el problema de clasificar entradas casi ordenadas. Argumente que el procedimiento CLASIFICACIÓN POR INSERCIÓN tendería a vencer al procedimiento CLASIFICACIÓN RÁPIDA en este problema.

#### 7.2-5

Suponga que las divisiones en cada nivel de clasificación rápida están en la proporción  $1, a, \dots$ , donde  $0 < a \leq 1$  es una constante. Muestre que la profundidad mínima de una hoja en el árbol de recurrencias es aproximadamente  $\lg n = \lg \frac{1}{a}$ , y la profundidad máxima es aproximadamente  $\lg n = \lg \frac{1}{a}$ . (No se preocupe por el redondeo de enteros).

## 7.2-6 ?

Argumente que para cualquier constante  $0 < c \leq 2$ , la probabilidad es de aproximadamente  $c/2$  de que en una matriz de entrada aleatoria, PARTITION produzca una división más equilibrada que  $1/c$ .

### 7.3 Una versión aleatoria de Quicksort

Al explorar el comportamiento del caso promedio de ordenación rápida, hemos asumido que todas las permutaciones de los números de entrada son igualmente probables. En una situación de ingeniería, sin embargo, no siempre podemos esperar que esta suposición se cumpla.

(Consulte el ejercicio 7.2-4.) Como vimos en la sección 5.3, a veces podemos agregar aleatorización a un algoritmo para obtener un buen desempeño esperado en todas las entradas. Mucha gente considera que la versión aleatoria resultante de quicksort es el algoritmo de clasificación elegido para entradas lo suficientemente grandes.

En la Sección 5.3, aleatorizamos nuestro algoritmo permutando explícitamente la entrada. Podríamos hacerlo también para la ordenación rápida, pero una técnica de aleatorización diferente, llamada muestreo aleatorio, produce un análisis más simple. En lugar de usar siempre  $A[i]$  como pivote, seleccionaremos un elemento elegido al azar del subarreglo  $A[p:r]$ . Lo hacemos intercambiando primero el elemento  $A[i]$  con un elemento elegido al azar de  $A[p:r]$ . Muestreando aleatoriamente el rango  $p : : ; r$ , aseguramos que el elemento pivote  $A[i]$  tiene la misma probabilidad de ser cualquiera de los elementos  $A[p:r]$ .

Debido a que elegimos aleatoriamente el elemento pivote, esperamos que la división de la matriz de entrada esté razonablemente bien equilibrada en promedio.

Los cambios en PARTITION y RANDOMIZED-PARTITION son pequeños. En la nueva partición procedimiento, simplemente implementamos el intercambio antes de particionar:

PARTICIÓN ALEATORIA.A; pag; r/ 1 i

D ALEATORIO.p; r/ 2

intercambia  $A[i]$  con  $A[r]$

regresa PARTICIÓN.A; pag; r/

El nuevo quicksort llama a RANDOMIZED-PARTITION en lugar de PARTITION:

RANDOMIZED-QUICKSORT.A; pag; r/

1 si  $p < r$  2 q

D PARTICIÓN ALEATORIA.A; pag; r/ 3 RANDOMIZED-

QUICKSORT.A; pag; q 1/ 4 RANDOMIZED-

QUICKSORT.A; q C 1; r/

Analizamos este algoritmo en la siguiente sección.

## Ejercicios

### 7.3-1

¿Por qué analizamos el tiempo de ejecución esperado de un algoritmo aleatorio y no su tiempo de ejecución en el peor de los casos?

### 7.3-2

Cuando se ejecuta RANDOMIZED-QUICKSORT , ¿cuántas llamadas se realizan al generador de números aleatorios RANDOM en el peor de los casos? ¿Qué tal en el mejor de los casos?

Da tu respuesta en términos de notación ,.

## 7.4 Análisis de clasificación rápida

La sección 7.2 proporcionó cierta intuición sobre el comportamiento en el peor de los casos de ordenación rápida y por qué esperamos que se ejecute rápidamente. En esta sección, analizamos el comportamiento de ordenación rápida con más rigor. Comenzamos con un análisis del peor de los casos, que se aplica a QUICKSORT o RANDOMIZED-QUICKSORT, y concluimos con un análisis del tiempo de ejecución esperado de RANDOMIZED-QUICKSORT.

### 7.4.1 Análisis del peor de los casos

Vimos en la Sección 7.2 que una división en el peor de los casos en cada nivel de recursividad en ordenación rápida produce un tiempo de ejecución  $,n^2/$ , que, intuitivamente, es el peor tiempo de ejecución del algoritmo. Probamos ahora esta afirmación.

Usando el método de sustitución (ver Sección 4.3), podemos mostrar que el tiempo de ejecución de quicksort es  $O.n^2/$ . Sea  $T .n/$  el tiempo en el peor de los casos para el procedimiento QUICKSORT en una entrada de tamaño  $n$ . Tenemos la recurrencia

$$T .n/ \leq D_{\max} + \sum_{0 \leq q \leq n-1} T .n-q/ + C .n/ ; \quad (7.1)$$

donde el parámetro  $q$  va de 0 a  $n-1$  porque el procedimiento PARTICIÓN produce dos subproblemas con tamaño total  $n-1$ . Suponemos que  $T .n/ \leq cn^2$  para alguna constante  $c$ . Sustituyendo esta conjectura en la recurrencia (7.1), obtenemos

$$T .n/ \leq D_{\max} + \sum_{0 \leq q \leq n-1} c(n-q)^2 C cn^2 / C .n/$$

$$D_{\max} + c \sum_{0 \leq q \leq n-1} (n-q)^2 C cn^2 / C .n/ :$$

La expresión  $c n^2 C .n/$  alcanza un máximo sobre el rango del parámetro  $0 \leq q \leq n-1$  en cualquier punto final. Para verificar esta afirmación, observe que la segunda derivada de la expresión con respecto a  $q$  es positiva (vea el ejercicio 7.4-3). Este

observación nos da el límite  $\max\{qn^1, qn^2\} \leq n^1/2 + n^2/2 \leq Cn^1$ . Continuando con nuestro límite de  $T(n)$ , obtenemos

$$T(n) \leq cn^2 + c_2 n^1/2 + cn^2$$

ya que podemos elegir la constante  $c$  lo suficientemente grande como para que el término  $c_2n^1/2$  domine al término  $c_2n^1$ . Así,  $T(n) \geq O(n^2)$ . Vimos en la Sección 7.2 un caso específico en el que quicksort toma  $n^2$  tiempo: cuando la partición está desequilibrada. Alternativamente, el Ejercicio 7.4-1 le pide que demuestre que la recurrencia (7.1) tiene una solución de  $T(n) = O(n^2)$ . Por lo tanto, el tiempo de ejecución (en el peor de los casos) de quicksort es  $O(n^2)$ .

#### 7.4.2 Tiempo de ejecución esperado

Ya hemos visto la intuición detrás de por qué el tiempo de ejecución esperado de RANDOMIZED-QUICKSORT es  $O(n \lg n)$ : si, en cada nivel de recursividad, la división inducida por RANDOMIZED-PARTITION pone cualquier fracción constante de los elementos en un lado de la partición, entonces el árbol de recurrencia tiene una profundidad  $\lg n$ , y el trabajo  $O(n)$  se realiza en cada nivel. Incluso si agregamos algunos niveles nuevos con la división más desequilibrada posible entre estos niveles, el tiempo total permanece en  $O(n \lg n)$ . Podemos analizar el tiempo de ejecución esperado de RANDOMIZED-QUICKSORT con precisión entendiendo primero cómo funciona el procedimiento de partición y luego usando esta comprensión para derivar un límite  $O(n \lg n)$  en el tiempo de ejecución esperado. Este límite superior del tiempo de ejecución esperado, combinado con el límite del mejor de los casos  $O(n \lg n)$  que vimos en la Sección 7.2, produce un tiempo de ejecución esperado de  $O(n \lg n)$ . Asumimos en todo momento que los valores de los elementos que se ordenan son distintos.

Tiempo de ejecución y comparaciones.

Los procedimientos QUICKSORT y RANDOMIZED-QUICKSORT difieren solo en cómo seleccionan los elementos pivote; son iguales en todos los demás aspectos. Por lo tanto, podemos formular nuestro análisis de RANDOMIZED-QUICKSORT discutiendo los procedimientos QUICKSORT y PARTITION, pero suponiendo que los elementos pivote se seleccionan aleatoriamente del subarreglo que se pasa a RANDOMIZED-PARTITION.

El tiempo de ejecución de QUICKSORT está dominado por el tiempo empleado en el procedimiento de PARTICIÓN. Cada vez que se llama al procedimiento PARTITION, selecciona un elemento pivote, y este elemento nunca se incluye en futuras llamadas recursivas a QUICKSORT y PARTITION. Por lo tanto, puede haber como máximo  $n$  llamadas a PARTICIÓN durante toda la ejecución del algoritmo de clasificación rápida. Una llamada a PARTICIÓN toma  $O(1)$  tiempo más una cantidad de tiempo que es proporcional al número de iteraciones del ciclo for en las líneas 3–6. Cada iteración de este ciclo for realiza una comparación en la línea 4, comparando el elemento pivote con otro elemento de la matriz  $A$ . Por lo tanto,

si podemos contar el número total de veces que se ejecuta la línea 4, podemos vincular el tiempo total pasado en el ciclo for durante toda la ejecución de QUICKSORT.

### Lema 7.1

Sea  $X$  el número de comparaciones realizadas en la línea 4 de PARTICIÓN sobre toda la ejecución de QUICKSORT en un arreglo de  $n$  elementos. Entonces el tiempo de ejecución de QUICKSORT es  $O(nX)$ .

Prueba Por la discusión anterior, el algoritmo realiza como máximo  $n$  llamadas a PARTITION, cada una de las cuales realiza una cantidad constante de trabajo y luego ejecuta el ciclo for varias veces. Cada iteración del ciclo for ejecuta la línea 4. ■

Nuestro objetivo, por lo tanto, es calcular  $X$ , el número total de comparaciones realizadas en todas las llamadas a PARTICIÓN. No intentaremos analizar cuántas comparaciones se realizan en cada llamada a PARTICIÓN. Más bien, derivaremos un límite general sobre el número total de comparaciones. Para hacerlo, debemos entender cuándo el algoritmo compara dos elementos de la matriz y cuándo no. Para facilitar el análisis, renombramos los elementos del arreglo A como ' $i$ ; ' $j$ ; ' $n$ ', siendo ' $i$ ' el  $i$ -ésimo elemento más pequeño. También definimos el conjunto  $Z_{ij}$  de los índices  $i$  y  $j$  para ser el conjunto de elementos entre ' $i$ ' y ' $j$ ', inclusive.

¿Cuándo compara el algoritmo ' $i$ ' y ' $j$ '? Para responder a esta pregunta, primero observamos que cada par de elementos se compara como máximo una vez. ¿Por qué? Los elementos se comparan solo con el elemento pivote  $y$ , después de que finaliza una llamada particular de PARTITION, el elemento pivote utilizado en esa llamada nunca más se compara con ningún otro elemento.

Nuestro análisis utiliza variables aleatorias indicadoras (consulte la Sección 5.2). Definimos

$X_{ij} = 1$  si se compara con ' $j$ ';

donde estamos considerando si la comparación tiene lugar en cualquier momento durante la ejecución del algoritmo, no solo durante una iteración o una llamada de PARTICIÓN.

Dado que cada par se compara como máximo una vez, podemos caracterizar fácilmente el número total de comparaciones realizadas por el algoritmo:

$$\sum_{i=1}^n \sum_{j=i+1}^n X_{ij}$$

Tomando las expectativas de ambos lados y luego usando la linealidad de la expectativa y el Lema 5.1, obtenemos

$$E[X_{ij}] = P(i < j) = \frac{1}{2}$$

$$\begin{matrix} Xn1 & Xn & E \infty Xij \\ \overline{iD1} & jDc1 \end{matrix}$$

$$\begin{matrix} Xn1 & Xn & Pr f'i se compara con 'j g : \\ \overline{iD1} & jDc1 \end{matrix} \quad (7.2)$$

Queda por calcular  $Pr f'i$  se compara con ' $j$  g. Nuestro análisis supone que el procedimiento de PARTICIÓN ALEATORIA elige cada pivote de forma aleatoria e independiente.

Pensemos en cuando dos elementos no se comparan. Considere una entrada para clasificar rápidamente los números del 1 al 10 (en cualquier orden) y suponga que el primer elemento pivote es 7. Luego, la primera llamada a PARTICIÓN separa los números en dos conjuntos:  $f1; 2; 3; 4; 5; 6g$  y  $f8; 9; 10g$ . Al hacerlo, el elemento de pivote 7 se compara con todos los demás elementos, pero ningún número del primer conjunto (p. ej., 2) se compara ni se comparará nunca con ningún número del segundo conjunto (p. ej., 9).

En general, debido a que asumimos que los valores de los elementos son distintos, una vez que se elige un pivote  $x$  con ' $i < x < j$ ', sabemos que ' $i$ ' y ' $j$ ' no se pueden comparar en ningún momento posterior. Si, por el contrario, se elige ' $i$ ' como pivote antes que cualquier otro elemento en  $Zij$ , entonces ' $i$ ' se comparará con cada elemento en  $Zij$ , excepto con él mismo. De manera similar, si ' $j$ ' se elige como pivote antes que cualquier otro elemento en  $Zij$ , entonces ' $j$ ' se comparará con cada elemento en  $Zij$ , excepto por sí mismo. En nuestro ejemplo, los valores 7 y 9 se comparan porque 7 es el primer elemento de  $Z7;9$  que se elige como pivote. Por el contrario, 2 y 9 nunca se compararán porque el primer elemento pivote elegido de  $Z2;9$  es 7. Por lo tanto, ' $i$ ' y ' $j$ ' se comparan si y solo si el primer elemento que se elige como pivote de  $Zij$  es ' $i$ ' o ' $j$ '.

Ahora calculamos la probabilidad de que ocurra este evento. Antes del punto en el que se ha elegido un elemento de  $Zij$  como pivote, todo el conjunto  $Zij$  se encuentra junto en la misma partición. Por lo tanto, es igualmente probable que cualquier elemento de  $Zij$  sea el primero elegido como pivote. Como el conjunto  $Zij$  tiene  $j - i + 1$  elementos, y como los pivotes se eligen aleatoriamente e independientemente, la probabilidad de que cualquier elemento dado sea el primero elegido como pivote es  $1/(j-i+1)$ . Así, tenemos

$Pr f'i$  se compara con ' $j$  g  $\leq Pr f'i$  o ' $j$ ' es el primer pivot elegido de  $Zij$  g

$D Pr f'i$  es el primer pivot elegido de  $Zij$  g

$C Pr f'j$  es el primer pivot elegido de  $Zij$  g

$$D \quad \frac{1}{j-i+2} \quad C \quad \frac{1}{j-i+1}$$

$$D \quad \frac{1}{j-i+1} \quad . \quad (7.3)$$

La segunda línea sigue porque los dos eventos son mutuamente excluyentes. Combinando las ecuaciones (7.2) y (7.3), obtenemos que

$$E \sum_{i \in D} \sum_{j \in C} X_{n1} X_n \frac{2}{j - C_1}$$

Podemos evaluar esta suma usando un cambio de variables ( $k = j - C_1$ ) y el límite de la serie armónica en la ecuación (A.7):

$$\begin{aligned} E \sum_{i \in D} \sum_{j \in C} X_{n1} X_n \frac{2}{j - C_1} &= \\ \sum_{i \in D} X_{n1} \frac{\sum_{k=1}^n \frac{2}{k - C_1}}{C_1} &= \\ < \sum_{i \in D} X_{n1} \frac{\sum_{k=1}^n \frac{2}{k}}{C_1} &= \\ X_{n1} \frac{O(\lg n)}{C_1} &= \\ D \text{ en } \lg n : & \end{aligned} \tag{7.4}$$

Por lo tanto, concluimos que, utilizando RANDOMIZED-PARTITION, el tiempo de ejecución esperado de quicksort es  $O(n \lg n)$  cuando los valores de los elementos son distintos.

### Ejercicios

#### 7.4-1

Muestre que en la recurrencia

$$T(n) \leq \max_{0 \leq q \leq 1} T(q) + CT(nq) + Cn$$

$$T(n) \leq nC$$

#### 7.4-2

Muestre que el mejor tiempo de ejecución de quicksort es  $n \lg n$ .

#### 7.4-3

Muestre que la expresión  $\frac{1}{2} \ln(1/q) + \frac{1}{2} \ln(n) + \dots + \ln(n)$  alcanza un máximo sobre  $q \in [0, 1]$  cuando  $q = 0$  o  $q = 1$ .

#### 7.4-4

Muestre que el tiempo de ejecución esperado de RANDOMIZED-QUICKSORT es  $n \lg n$ .

## 7.4-5

Podemos mejorar el tiempo de ejecución de la ordenación rápida en la práctica aprovechando el rápido tiempo de ejecución de la ordenación por inserción cuando su entrada está "casi" ordenada. Al llamar a quicksort en un subarreglo con menos de  $k$  elementos, simplemente regrese sin ordenar el subarreglo. Después de que regrese la llamada de nivel superior a Quicksort, ejecute la ordenación por inserción en toda la matriz para finalizar el proceso de ordenación. Argumente que este algoritmo de clasificación se ejecuta en  $O(nk \lg n + k^2)$  tiempo esperado. ¿Cómo debemos elegir  $k$ , tanto en la teoría como en la práctica?

## 7.4-6 ?

Considere la posibilidad de modificar el procedimiento de PARTICIÓN seleccionando al azar tres elementos de la matriz  $A$  y dividiendo sobre su mediana (el valor medio de los tres elementos). Calcule la probabilidad de obtener, en el peor de los casos, una división de  $A$  en dos partes de  $\lfloor \frac{1}{3} \rfloor$  y  $\lceil \frac{2}{3} \rceil$ , como una función de  $n$  en el rango  $0 < \epsilon < 1$ .

## Problemas

## 7-1 Corrección de la partición de Hoare

La versión de PARTICIÓN dada en este capítulo no es el algoritmo de partición original. Aquí está el algoritmo de partición original, que se debe a CAR Hoare:

HOARE-PARTICIÓN.A; pag; r/ 1 x

```

D AŒp 2 i D p
1 3 j D r C 1 4
mientras TRUE 5
    repetir 6 j D j 7
        hasta AŒj 8 repetir
            9 i D i C 1 10 hasta AŒi      1
            x 11 si i<j 12           X
            intercambiar ACEi
            con AŒj 13 más volver j

```

- Demostrar el funcionamiento de HOARE-PARTITION en el arreglo  $A[1..13] = [19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]$ , que muestra los valores de la matriz y los valores auxiliares después de cada iteración del ciclo while en las líneas 4–13.

Las siguientes tres preguntas le piden que dé un argumento cuidadoso de que el procedimiento HOARE-PARTITION es correcto. Suponiendo que el subarreglo  $A[ep:r]$  contiene al menos dos elementos, pruebe lo siguiente:

- b. Los índices  $i$  y  $j$  son tales que nunca accedemos a un elemento de  $A$  fuera del subarreglo  $A[ep:r]$ .
- C. Cuando termina HOARE-PARTITION, devuelve un valor  $j$  tal que  $pj < r$ .
- d. Cada elemento de  $A[ep:j]$  es menor o igual que cada elemento de  $A[j:r]$  cuando termina HOARE-PARTITION.

El procedimiento de PARTICIÓN en la Sección 7.1 separa el valor pivote (originalmente en  $A[er]$ ) de las dos particiones que forma. El procedimiento HOARE-PARTITION, por otro lado, siempre coloca el valor pivote (originalmente en  $A[ep]$ ) en una de las dos particiones  $A[ep:j]$  y  $A[j:r]$ . Dado que  $pj < r$ , esta división siempre es no trivial.

mi. Vuelva a escribir el procedimiento QUICKSORT para usar HOARE-PARTITION.

#### 7-2 Ordenación rápida con valores de elementos

iguales El análisis del tiempo de ejecución esperado de la ordenación rápida aleatoria en la Sección 7.4.2 supone que todos los valores de los elementos son distintos. En este problema, examinamos qué sucede cuando no lo son.

- a. Suponga que todos los valores de los elementos son iguales. ¿Cuál sería el tiempo de ejecución de la ordenación rápida aleatoria en este caso?
- b. El procedimiento PARTITION devuelve un índice  $q$  tal que cada elemento de  $A[q:1]$  es menor o igual que  $A[q]$  y cada elemento de  $A[q+1:r]$  es mayor que  $A[q]$ . Modifique el procedimiento PARTITION para producir un procedimiento PARTITION0 .A; pag; r/, que permuta los elementos de  $A[ep:r]$  y devuelve dos índices  $q$  y  $t$ , donde  $pqr$ , tal que

todos los elementos de  $A[q:t]$  son  
iguales, cada elemento de  $A[q+1:t]$  es menor que  
 $A[q]$ , y cada elemento de  $A[t:r]$  es mayor que  $A[q]$ .

Al igual que PARTITION, su procedimiento PARTITION0 debería tomar ,rp/ tiempo.

- C. Modifique el procedimiento RANDOMIZED-QUICKSORT para llamar a PARTITION0, y nombre el nuevo procedimiento RANDOMIZED-QUICKSORT0 . Luego modifique el procedimiento QUICKSORT para producir un procedimiento QUICKSORT0 .p; r/ eso llama

RANDOMIZED-PARTITION0 y se repite solo en particiones de elementos que no se sabe que son iguales entre sí.

- d. Usando QUICKSORT0 , ¿cómo ajustaría el análisis en la Sección 7.4.2 para evitar la suposición de que todos los elementos son distintos?

### 7-3 Análisis alternativo de ordenación

rápida Un análisis alternativo del tiempo de ejecución de la ordenación rápida aleatoria se centra en el tiempo de ejecución esperado de cada llamada recursiva individual a ORDENACIÓN RÁPIDA ALEATORIA, en lugar del número de comparaciones realizadas.

- a. Argumente que, dado un arreglo de tamaño n, la probabilidad de que cualquier elemento en particular sea elegido como pivote es  $1/n$ . Úselo para definir variables aleatorias indicadoras.  $X_i$  si se elige el elemento más pequeño como pivote. ¿Qué es  $E \sum X_i$  ?
- b. Sea  $T(n)$  una variable aleatoria que indica el tiempo de ejecución de quicksort en una matriz de tamaño n. Argumenta eso

(7.5)

$$E \sum T(k) = \sum_{k=1}^n k T(k) = \sum_{k=1}^n k \cdot \frac{1}{n} \sum_{i=1}^n T(i) = \frac{1}{n} \sum_{i=1}^n i T(i) = \frac{1}{n} \sum_{i=1}^n i \cdot \frac{1}{n} \sum_{j=1}^i T(j) = \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^i T(j)$$

- c. Demuestre que podemos reescribir la ecuación (7.5) como

$$E \sum T(k) = \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^i T(j) = \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^i \left( \frac{1}{n} \sum_{k=1}^j T(k) \right) = \frac{1}{n^3} \sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j T(k) = \frac{1}{n^3} \sum_{k=1}^n k T(k) = \frac{1}{n^3} \sum_{k=1}^n k \cdot \frac{1}{n} \sum_{i=1}^k T(i) = \frac{1}{n^4} \sum_{k=1}^n \sum_{i=1}^k T(i)$$

- d. Muestra esa

$$\frac{1}{kD2} \sum_{k=1}^n k \lg k = \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 + \dots$$

(Sugerencia: divida la sumatoria en dos partes, una para  $k \leq D/2$ ; otra para  $k > D/2$ .)

- mi. Usando el límite de la ecuación (7.7), demuestre que la recurrencia en la ecuación (7.6) tiene la solución  $E \sum T(k) = n \lg n$ . (Sugerencia: demuestre, por sustitución, que  $E \sum T(k) \leq an \lg n$  para  $n$  suficientemente grande y para alguna constante positiva a).

#### 7-4 Profundidad de pila para clasificación

rápida El algoritmo QUICKSORT de la Sección 7.1 contiene dos llamadas recursivas a sí mismo. Después de que QUICKSORT llama a PARTITION, ordena recursivamente el subarreglo izquierdo y luego ordena recursivamente el subarreglo derecho. La segunda llamada recursiva en QUICKSORT no es realmente necesaria; podemos evitarlo usando una estructura de control iterativa. Esta técnica, llamada recursión de cola, es proporcionada automáticamente por buenos compiladores. Considere la siguiente versión de ordenación rápida, que simula la recursividad de la cola:

```
COLA-RECUSIVO-QUICKSORT.A; pag; r/ 1
while p<r 2 //
Particionar y ordenar el subarreglo izquierdo. 3 q D
TABIQUE.A; pag; r/ 4 COLA-RECUSIVO-
QUICKSORT.A; pag; q 1/ 5 pag re q do 1
```

- a. Argumente que TAIL-RECUSIVE-QUICKSORT.A; 1; A:longitud/ ordena correctamente el matriz a

Los compiladores normalmente ejecutan procedimientos recursivos usando una pila que contiene información pertinente, incluidos los valores de los parámetros, para cada llamada recursiva. La información de la llamada más reciente está en la parte superior de la pila y la información de la llamada inicial está en la parte inferior. Al llamar a un procedimiento, su información se coloca en la pila; cuando termina, su información aparece. Dado que asumimos que los parámetros de la matriz están representados por punteros, la información para cada llamada de procedimiento en la pila requiere  $O(1)$  espacio de pila. La profundidad de pila es la cantidad máxima de espacio de pila utilizada en cualquier momento durante un cálculo.

- b. Describa un escenario en el que la profundidad de pila de TAIL-RECUSIVE-QUICKSORT sea  $,n$  en una matriz de entrada de  $n$  elementos.

- C. Modifique el código para TAIL-RECUSIVE-QUICKSORT para que la profundidad de pila en el peor de los casos sea  $,\lg n$ . Mantenga el tiempo de ejecución esperado  $O(\lg n)$  del algoritmo.

#### 7-5 Partición de mediana de 3 Una

forma de mejorar el procedimiento RANDOMIZED-QUICKSORT es particionar alrededor de un pivote que se elige con más cuidado que seleccionando un elemento aleatorio del subarreglo. Un enfoque común es el método de la mediana de 3 : elija el pivote como la mediana (elemento central) de un conjunto de 3 elementos seleccionados al azar del subarreglo. (Vea el ejercicio 7.4-6.) Para este problema, supongamos que los elementos en el arreglo de entrada  $A[1:n]$  son distintos y que  $n \geq 3$ . Denotamos el

matriz de salida ordenada por  $A[0 : : n]$ . Usando el método de la mediana de 3 para elegir el elemento pivote  $x$ , defina  $p_i$  D  $\Pr_{x \in A[i : : n]}{x = p_i}$ .

- a. Dé una fórmula exacta para  $p_i$  en función de  $n$  para  $i \in \{1, 2, \dots, n\}$ .  
(Observe que  $p_1 = \frac{1}{n}$ .)
- b. ¿En qué cantidad hemos aumentado la probabilidad de elegir el pivote como  $x \in A[0 : : n]$  si  $c = 2$ , la mediana de  $A[0 : : n]$ , en comparación con la implementación ordinaria? Suponga que  $n \geq 1$ , y dé la relación límite de estas probabilidades.
- C. Si definimos una "buena" división para significar elegir el pivote como  $x \in A[0 : : i]$ , donde  $i = \lceil \frac{n}{3} \rceil$ , ¿en qué cantidad hemos aumentado la probabilidad de obtener una buena división en comparación con la implementación ordinaria? (Sugerencia: aproxime la suma mediante una integral).
- d. Argumente que en el tiempo de ejecución  $\Theta(n \lg n)$  de quicksort, el método de la mediana de 3 afecta solo al factor constante.

#### 7-6 Clasificación difusa de intervalos

Considere un problema de clasificación en el que no conocemos los números exactamente. En cambio, para cada número conocemos un intervalo en la recta real a la que pertenece. Es decir, se nos dan  $n$  intervalos cerrados de la forma  $[a_i, b_i]$ , donde  $a_i < b_i$ . Deseamos ordenar estos intervalos de forma difusa, es decir, producir una permutación  $i_1, i_2, \dots, i_n$  de los intervalos tales que para  $j \in \{1, 2, \dots, n\}$ , existen  $c_j$  tales que  $a_{i_{c_j}} \leq b_{i_{c_j}}$ .

$c_1, c_2, \dots, c_n$

- a. Diseñe un algoritmo aleatorio para la clasificación difusa de  $n$  intervalos. Su algoritmo debe tener la estructura general de un algoritmo que clasifica rápidamente los puntos extremos izquierdos (los valores de  $a_i$ ), pero debe aprovechar los intervalos superpuestos para mejorar el tiempo de ejecución. (A medida que los intervalos se superponen cada vez más, el problema de la clasificación difusa de los intervalos se vuelve cada vez más fácil. Su algoritmo debe aprovechar dicha superposición, en la medida en que exista).
- b. Argumente que su algoritmo se ejecuta en el tiempo esperado  $\Theta(n \lg n)$  en general, pero se ejecuta en el tiempo esperado  $\Theta(n^2)$  cuando todos los intervalos se superponen (es decir, cuando existe un valor  $x$  tal que  $x \in [a_i, b_i]$  para todos  $i$ ). Su algoritmo no debería verificar este caso explícitamente; más bien, su rendimiento debería mejorar naturalmente a medida que aumenta la cantidad de superposición.

### Notas del capítulo

El procedimiento quicksort fue inventado por Hoare [170]; La versión de Hoare aparece en el Problema 7-1. El procedimiento de PARTICIÓN dado en la Sección 7.1 se debe a N. Lomuto. El análisis de la Sección 7.4 se debe a Avrim Blum, Sedgewick [305] y Bentley [43] brindan una buena referencia sobre los detalles de la implementación y cómo asunto.

McIlroy [248] mostró cómo diseñar un "adversario asesino" que produce una matriz en la que prácticamente cualquier implementación de clasificación rápida toma  $n^2$  tiempo. Si la implementación es aleatoria, el adversario produce la matriz después de ver las opciones aleatorias del algoritmo de clasificación rápida.

## 8

## Clasificación en tiempo lineal

Ahora hemos introducido varios algoritmos que pueden clasificar  $n$  números en  $O(n \lg n)$  time. Merge sort y heapsort logran este límite superior en el peor de los casos; quicksort lo logra en promedio. Además, para cada uno de estos algoritmos, podemos producir una secuencia de  $n$  números de entrada que hace que el algoritmo se ejecute en  $\Omega(n^2)$  tiempo.

Estos algoritmos comparten una propiedad interesante: el orden ordenado que determinan se basa únicamente en comparaciones entre los elementos de entrada. A estos algoritmos de clasificación los llamamos tipos de comparación. Todos los algoritmos de clasificación presentados hasta ahora son clasificaciones de comparación.

En la Sección 8.1, probaremos que cualquier clasificación por comparación debe hacer  $\Omega(n \lg n)$  comparaciones en el peor de los casos para clasificar  $n$  elementos. Por lo tanto, la ordenación por fusión y la ordenación por montón son asintóticamente óptimas, y no existe una ordenación por comparación que sea más rápida por más de un factor constante.

Las secciones 8.2, 8.3 y 8.4 examinan tres algoritmos de clasificación (clasificación por conteo, clasificación por base y clasificación por cubo) que se ejecutan en tiempo lineal. Por supuesto, estos algoritmos utilizan operaciones distintas de las comparaciones para determinar el orden de clasificación. En consecuencia, el límite inferior  $\Omega(n \lg n)$  no se aplica a ellos.

### 8.1 Límites inferiores para la clasificación

En una ordenación por comparación, usamos solo comparaciones entre elementos para obtener información de orden sobre una secuencia de entrada  $a_1, a_2, \dots, a_n$ . Es decir, dados dos elementos  $a_i$  y  $a_j$ , realizamos una de las pruebas  $a_i < a_j$ ,  $a_i = a_j$  o  $a_i > a_j$  para determinar su orden relativo. No podemos inspeccionar los valores de los elementos ni obtener información sobre ellos de ningún otro modo.

En esta sección, asumimos sin pérdida de generalidad que todos los elementos de entrada son distintos. Dada esta suposición, las comparaciones de la forma  $a_i = a_j$  son inútiles, por lo que podemos suponer que no se realizan comparaciones de esta forma. También notamos que las comparaciones  $a_i < a_j$ ,  $a_i = a_j$  y  $a_i > a_j$  son todas equivalentes en que

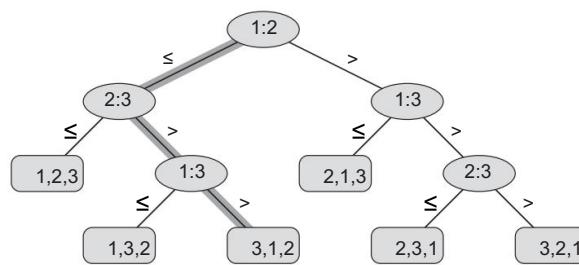


Figura 8.1 El árbol de decisión para la ordenación por inserción que opera en tres elementos. Un nodo interno anotado por  $i:j$  indica una comparación entre  $a_i$  y  $a_j$ . Una hoja anotada por la permutación  $h.1/; .2/; \dots; .n/i$  indica el orden  $a_1/a_2/\dots/a_n$ . La ruta sombreada indica las decisiones tomadas al ordenar la secuencia de entrada ha1 D 6; a2 D 8; a3 D 5; la permutación h3; 1; 2 i en la hoja indica que el orden ordenado es a3 D 5 a1 D 6 a2 D 8. Hay  $3^3 = 27$  permutaciones posibles de los elementos de entrada, por lo que el árbol de decisión debe tener al menos 6 hojas.

dan información idéntica sobre el orden relativo de  $a_i$  y  $a_j$ . Por lo tanto, asumimos que todas las comparaciones tienen la forma  $a_i a_j$ .

#### El modelo de árbol de decisión

Podemos ver las clasificaciones de comparación de manera abstracta en términos de árboles de decisión. Un árbol de decisión es un árbol binario completo que representa las comparaciones entre elementos realizadas por un algoritmo de clasificación particular que opera en una entrada de un tamaño determinado. Se ignoran el control, el movimiento de datos y todos los demás aspectos del algoritmo. La Figura 8.1 muestra el árbol de decisión correspondiente al algoritmo de clasificación por inserción de la Sección 2.1 que opera en una secuencia de entrada de tres elementos.

En un árbol de decisión, anotamos cada nodo interno por  $i:j$  para algunos  $i$  y  $j$  en el rango  $1 \leq i, j \leq n$ , donde  $n$  es el número de elementos en la secuencia de entrada. También anotamos cada hoja por una permutación  $h.1/; .2/; \dots; .n/i$ . (Consulte la Sección C.1 para conocer los antecedentes de las permutaciones.) La ejecución del algoritmo de clasificación corresponde a trazar un camino simple desde la raíz del árbol de decisión hasta una hoja.

Cada nodo interno indica una comparación  $a_i a_j$ . El subárbol izquierdo dicta las comparaciones posteriores una vez que sabemos que  $a_i a_j$ , y el subárbol derecho dicta las comparaciones posteriores sabiendo que  $a_i > a_j$ . Cuando llegamos a una hoja, el algoritmo de ordenamiento ha establecido el ordenamiento  $a_1/a_2/\dots/a_n$ . Debido a que cualquier algoritmo de clasificación correcto debe ser capaz de producir cada permutación de su entrada, cada una de las  $n!$  permutaciones en  $n$  elementos debe aparecer como una de las hojas del árbol de decisión para que una comparación de clasificación sea correcta. Además, cada una de estas hojas debe ser accesible desde la raíz por un camino descendente correspondiente a un

ejecución del género de comparación. (Nos referiremos a tales hojas como "alcanzables"). Por lo tanto, consideraremos solo árboles de decisión en los que cada permutación aparece como una hoja alcanzable.

#### Un límite inferior para el peor de los casos

La longitud del camino simple más largo desde la raíz de un árbol de decisión hasta cualquiera de sus hojas alcanzables representa el número de comparaciones en el peor de los casos que realiza el algoritmo de clasificación correspondiente. En consecuencia, el número de comparaciones en el peor de los casos para un algoritmo de clasificación de comparación dado es igual a la altura de su árbol de decisión. Un límite inferior en las alturas de todos los árboles de decisión en los que cada permutación aparece como una hoja alcanzable es, por lo tanto, un límite inferior en el tiempo de ejecución de cualquier algoritmo de clasificación por comparación. El siguiente teorema establece dicho límite inferior.

#### Teorema 8.1

Cualquier algoritmo de clasificación por comparación requiere comparaciones  $n \lg n$  en el peor de los casos.

Prueba De la discusión anterior, es suficiente para determinar la altura de un árbol de decisión en el que cada permutación aparece como una hoja alcanzable. Considere un árbol de decisión de altura  $h$  con  $I$  hojas alcanzables correspondientes a una ordenación de comparación en  $n$  elementos. Debido a que cada una de las permutaciones  $n!$  de la entrada aparece como una hoja, tenemos  $n! I$ . Como un árbol binario de altura  $h$  no tiene más de  $2h$  hojas, tenemos

$$n! I \leq 2^h$$

:

lo que, tomando logaritmos, implica

$$h \geq \lg(n!) / (\text{ya que la función } \lg \text{ aumenta monótonamente})$$

$$D \cdot n \lg n / (\text{por la ecuación (3.19)}) .$$

■

#### Corolario 8.2

Heapsort y merge sort son ordenaciones de comparación asintóticamente óptimas.

Prueba Los límites superiores de  $O(n \lg n)$  en los tiempos de ejecución para heapsort y merge sort coinciden con el límite inferior de  $n \lg n$  en el peor de los casos del Teorema 8.1. ■

#### Ejercicios

##### 8.1-1

¿Cuál es la menor profundidad posible de una hoja en un árbol de decisión para una ordenación de comparación?

## 8.1-2

Obtener límites ajustados asintóticamente en  $\lg n \tilde{S}$  sin utilizar la aproximación de Stirling. En cambio, evalúe la suma  $\sum_{k=1}^n \lg k$  usando técnicas de Sec  
ción A.2.

KD1

## 8.1-3

Demuestre que no existe una ordenación de comparación cuyo tiempo de ejecución sea lineal para al menos la mitad de las  $n \tilde{S}$  entradas de longitud  $n$ . ¿Qué pasa con una fracción de  $1=n$  de las entradas de longitud  $n$ ? ¿Qué pasa con una fracción  $1=2n$ ?

## 8.1-4

Suponga que le dan una secuencia de  $n$  elementos para ordenar. La secuencia de entrada consta de  $n=k$  subsecuencias, cada una de las cuales contiene  $k$  elementos. Los elementos de una subsecuencia dada son todos más pequeños que los elementos de la subsecuencia siguiente y más grandes que los elementos de la subsecuencia anterior. Por lo tanto, todo lo que se necesita para clasificar toda la secuencia de longitud  $n$  es clasificar los  $k$  elementos en cada una de las  $n=k$  subsecuencias. Muestre un límite inferior  $\Omega(n \lg k)$  en el número de comparaciones necesarias para resolver esta variante del problema de clasificación. (Sugerencia: no es riguroso combinar simplemente los límites inferiores de las subsecuencias individuales).

## 8.2 Clasificación de conteo

La ordenación por conteo asume que cada uno de los  $n$  elementos de entrada es un número entero en el rango de 0 a  $k$ , para algún número entero  $k$ . Cuando  $k \leq n$ , la ordenación se ejecuta en tiempo  $\Theta(n)$ .

La ordenación por conteo determina, para cada elemento de entrada  $x$ , el número de elementos menor que  $x$ . Utiliza esta información para colocar el elemento  $x$  directamente en su posición en la matriz de salida. Por ejemplo, si 17 elementos son menores que  $x$ , entonces  $x$  pertenece a la posición de salida 18. Debemos modificar este esquema ligeramente para manejar la situación en la que varios elementos tienen el mismo valor, ya que no queremos ponerlos todos en el mismo valor. posición.

En el código para ordenar por conteo, asumimos que la entrada es una matriz  $A[0:n-1]$ , y por lo tanto  $A:\text{longitud } D = n$ . Necesitamos otros dos arreglos: el arreglo  $B[0:n-1]$  contiene la salida ordenada, y el arreglo  $C[0:k-1]$  proporciona almacenamiento de trabajo temporal.

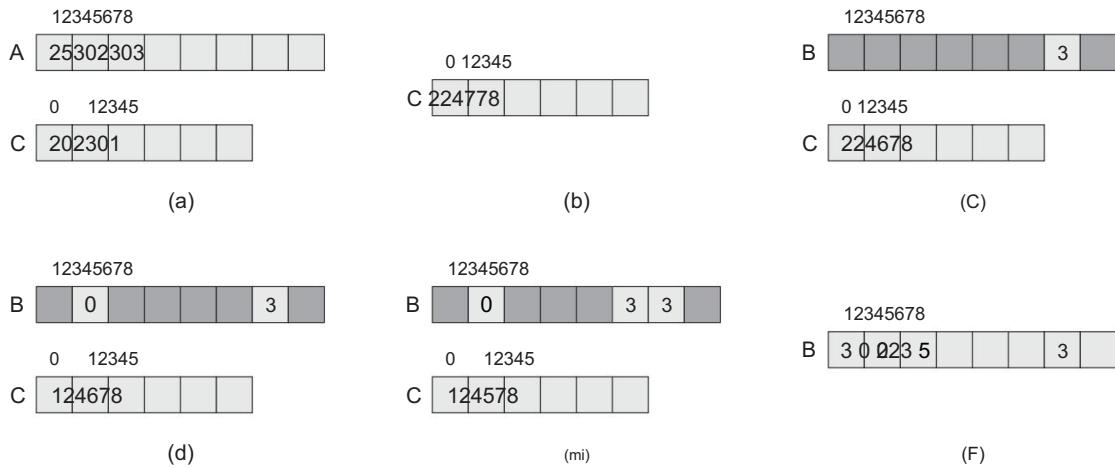


Figura 8.2 La operación de COUNTING-SORT en un arreglo de entrada  $A[1 : : 8]$ , donde cada elemento de A es un número entero no negativo no mayor que  $k = 5$ . (a) El arreglo A y el arreglo auxiliar C después de la línea 5. (b) El arreglo C después de la línea 8. (c)–(e) El arreglo de salida B y el arreglo auxiliar C después de una, dos y tres iteraciones del ciclo en las líneas 10–12, respectivamente. Solo se han rellenado los elementos ligeramente sombreados de la matriz B. (f) La matriz B de salida ordenada final.

CONTADOR-SORT.A; B; k / 1

```

sea C[0 : : k] un nuevo arreglo 2
para i D 0 a k C[i] D
    3          0 4 para j
    D 1 a A:longitud C[A][j] D
        C[j] C[j] + 1 // C[i] ahora
    contiene el número de elementos igual a i . 7 para i D 1 a k C[i] D C[i]
    C[i] 1

```

```

8 9 // C[i] ahora contiene el número de elementos menor o igual que i. 10 para j D
A:longitud hasta 1 11 B[C[i]] D
A[i] 12 C[A][j] D C[A][j]

```

1

La figura 8.2 ilustra la clasificación por conteo. Después de que el ciclo for de las líneas 2 y 3 inicialice el arreglo C a todos ceros, el ciclo for de las líneas 4 y 5 inspecciona cada elemento de entrada. Si el valor de un elemento de entrada es  $i$ , incrementamos  $C[i]$ . Así, después de la línea 5,  $C[i]$  contiene el número de elementos de entrada igual a  $i$  para cada entero  $i \in \{0, 1, \dots, k\}$ . Las líneas 7 y 8 determinan para cada  $i \in \{0, 1, \dots, k\}$  cuántos elementos de entrada son menores o iguales a  $i$  manteniendo una suma continua de la matriz C.

Finalmente, el ciclo for de las líneas 10–12 coloca cada elemento  $A[i]$  en su posición ordenada correcta en la matriz de salida  $B$ . Si todos los  $n$  elementos son distintos, entonces cuando ingresamos por primera vez la línea 10, para cada  $A[i]$ , el valor  $C[i][A[i]]$  es el correcto. posición final de  $A[i]$  en la matriz de salida, ya que hay elementos  $C[i][A[i]]$  menores o iguales que  $A[i]$ . Debido a que los elementos pueden no ser distintos, decrementamos  $C[i][A[i]]$  cada vez que colocamos un valor  $A[i]$  en la matriz  $B$ . La disminución de  $C[i][A[i]]$  hace que el siguiente elemento de entrada con un valor igual a  $A[i]$ , si existe, vaya a la posición inmediatamente anterior a  $A[i]$  en la matriz de salida.

¿Cuánto tiempo requiere ordenar el conteo? El ciclo for de las líneas 2–3 toma un tiempo  $,k$ , el ciclo for de las líneas 4–5 toma un tiempo  $,n$ , el ciclo for de las líneas 7–8 toma un tiempo  $,k$ , y el ciclo for de las líneas 10–12 toman el tiempo  $,n$ . Así, el tiempo total es  $,k + n$ . En la práctica, solemos usar la ordenación por conteo cuando tenemos  $k \leq n$ , en cuyo caso el tiempo de ejecución es  $,n$ .

La ordenación por conteo supera el límite inferior de  $n \lg n$  demostrado en la Sección 8.1 porque no es una ordenación por comparación. De hecho, no se producen comparaciones entre los elementos de entrada en ninguna parte del código. En cambio, la ordenación por conteo utiliza los valores reales de los elementos para indexarlos en una matriz. El límite inferior  $n \lg n$  para la clasificación no se aplica cuando nos apartamos del modelo de clasificación de comparación.

Una propiedad importante del ordenamiento por conteo es que es estable: los números con el mismo valor aparecen en la matriz de salida en el mismo orden que en la matriz de entrada. Es decir, rompe los lazos entre dos números por la regla de que el número que aparece primero en la matriz de entrada aparece primero en la matriz de salida. Normalmente, la propiedad de estabilidad es importante solo cuando los datos del satélite se transportan con el elemento que se está clasificando. La estabilidad de la ordenación por conteo es importante por otra razón: la ordenación por conteo a menudo se usa como una subrutina en la ordenación por raíz. Como veremos en la siguiente sección, para que la ordenación basada en radix funcione correctamente, la ordenación por conteo debe ser estable.

## Ejercicios

### 8.2-1

Utilizando la figura 8.2 como modelo, ilustre la operación de CONTAR-CLASIFICAR en el arreglo  $AD[0..6] = [0; 2; 0; 1; 3; 4; 6; 1; 3; 2]$ .

### 8.2-2

Demuestre que COUNTING-SORT es estable.

### 8.2-3

Supongamos que tuviéramos que reescribir el encabezado del bucle for en la línea 10 de COUNTING SORT como

`10 para j D 1 a A:longitud`

Muestre que el algoritmo aún funciona correctamente. ¿Es estable el algoritmo modificado?

### 8.2-4

Describa un algoritmo que, dados  $n$  enteros en el rango de 0 a  $k$ , preprocesa su entrada y luego responde cualquier consulta sobre cuántos de los  $n$  enteros caen en un rango  $a : : b$  en  $O(1/k)$  tiempo. Su algoritmo debe usar  $O(n \log k)$  tiempo de preprocesamiento.

---

### 8.3 Clasificación de raíz

Radix sort es el algoritmo utilizado por las máquinas clasificadoras de tarjetas que ahora solo se encuentran en los museos de computadoras. Las tarjetas tienen 80 columnas, y en cada columna una máquina puede hacer un agujero en uno de los 12 lugares. El clasificador se puede "programar" mecánicamente para examinar una columna determinada de cada tarjeta en una baraja y distribuir la tarjeta en uno de los 12 contenedores, según el lugar en el que se haya perforado. Entonces, un operador puede juntar las tarjetas caja por caja, de modo que las tarjetas con el primer lugar perforado estén encima de las tarjetas con el segundo lugar perforado, y así sucesivamente.

Para dígitos decimales, cada columna usa solo 10 lugares. (Los otros dos lugares están reservados para codificar caracteres no numéricos). Un número de  $d$  dígitos ocuparía entonces un campo de  $d$  columnas. Dado que el clasificador de tarjetas puede mirar solo una columna a la vez, el problema de clasificar  $n$  tarjetas en un número de  $d$  dígitos requiere un algoritmo de clasificación.

Intuitivamente, puede ordenar los números en su dígito más significativo, ordenar cada uno de los contenedores resultantes de forma recursiva y luego combinar los mazos en orden. Desafortunadamente, dado que las tarjetas en 9 de los 10 contenedores deben reservarse para clasificar cada uno de los contenedores, este procedimiento genera muchas pilas intermedias de tarjetas de las que tendrá que hacer un seguimiento. (Consulte el ejercicio 8.3-5.)

Radix sort resuelve el problema de la clasificación de tarjetas, de manera contraria a la intuición, al clasificar primero el dígito menos significativo. Luego, el algoritmo combina las cartas en una sola baraja, con las cartas en el contenedor 0 antes que las cartas en el contenedor 1 que preceden a las cartas en el contenedor 2, y así sucesivamente. Luego, vuelve a ordenar todo el mazo por el segundo dígito menos significativo y vuelve a combinar el mazo de la misma manera. El proceso continúa hasta que las tarjetas se han clasificado en todos los dígitos  $d$ . Sorprendentemente, en ese momento las tarjetas están completamente clasificadas en el número de dígito  $d$ . Por lo tanto, solo se requieren  $d$  pasos a través de la plataforma para clasificar. La figura 8.3 muestra cómo opera la ordenación radix en un "mazo" de siete números de 3 dígitos.

Para que la ordenación radix funcione correctamente, la ordenación por dígitos debe ser estable. La clasificación que realiza un clasificador de tarjetas es estable, pero el operador debe tener cuidado de no cambiar el orden de las tarjetas a la salida de un contenedor, aunque todas las tarjetas de un contenedor tengan el mismo dígito en la columna elegida.

329	720	720	329
457	355	329	355
657	436	436	436
839	.....; »	457	.....; »
436	657	355	657
720	329	457	720
355	839	657	839

Figura 8.3 La operación de ordenar por raíz en una lista de siete números de 3 dígitos. La columna más a la izquierda es la entrada. Las columnas restantes muestran la lista después de ordenaciones sucesivas en posiciones de dígitos cada vez más significativas. El sombreado indica la posición del dígito ordenada para producir cada lista a partir de la anterior.

En una computadora típica, que es una máquina secuencial de acceso aleatorio, algunas veces usamos la ordenación radix para ordenar registros de información que están codificados por múltiples campos. Por ejemplo, podríamos querer ordenar las fechas por tres claves: año, mes y día. Podríamos ejecutar un algoritmo de clasificación con una función de comparación que, dadas dos fechas, compare años, y si hay un empate, compare meses, y si ocurre otro empate, compare días. Alternativamente, podríamos clasificar la información tres veces con una clasificación estable: primero por día, luego por mes y finalmente por año.

El código para ordenar radix es sencillo. El siguiente procedimiento asume que cada elemento en la matriz A de n elementos tiene d dígitos, donde el dígito 1 es el dígito de orden más bajo y el dígito d es el dígito de orden más alto.

```
RADIX-SORT.A; d / 1
para i D 1 a d use una
2      ordenación estable para ordenar la matriz A en el dígito i
```

### Lema 8.3

Dados números de n d dígitos en los que cada dígito puede tomar hasta k valores posibles, RADIX-SORT ordena correctamente estos números en  $,dn C k//$  tiempo si la ordenación estable que usa toma  $,n C k//$  tiempo.

Prueba La corrección de la ordenación radix se obtiene por inducción en la columna que se está ordenando (vea el Ejercicio 8.3-3). El análisis del tiempo de ejecución depende de la clasificación estable utilizada como algoritmo de clasificación intermedio. Cuando cada dígito está en el rango de 0 a  $k-1$  (para que pueda tomar k valores posibles), y k no es demasiado grande, la clasificación por conteo es la opción obvia. Cada paso sobre números de n d dígitos lleva un tiempo  $,n C k//$ .

Hay d pasos, por lo que el tiempo total para la ordenación por raíz es  $,dn C k//$ . ■

Cuando d es constante y k D On/, podemos hacer que radix sort se ejecute en tiempo lineal. De manera más general, tenemos cierta flexibilidad en cómo dividir cada clave en dígitos.

**Lema 8.4**

Dados  $n$  números de  $b$  bits y cualquier entero positivo  $r_b$ , RADIX-SORT ordena correctamente estos números en  $\dots b=r/n C 2r //$  tiempo si la ordenación estable que usa toma  $.n C k /$  tiempo para entradas en el rango de 0 a  $k$ .

Para un valor  $r$  cada uno.  $b$ , consideramos que cada clave tiene  $d$   $db=re$  dígitos de  $r$  bits Prueba Cada dígito es un número entero en el rango de 0 a  $2r-1$ , de modo que podemos usar la ordenación por conteo con  $k = D 2r-1$ . (Por ejemplo, podemos ver una palabra de 32 bits como si tuviera cuatro dígitos de 8 bits, de modo que  $b = D 32$ ,  $r = D 8$ ,  $k = D 2r-1 = D 255$ , y  $d = D b=r = D 4$ .) Cada pase de tipo de conteo toma tiempo  $.n C k / D$ ,  $.n C 2r /$  y hay  $d$  pasos, por un tiempo de funcionamiento total de  $.dn C 2r // D$ ,  $\dots b=r/n C 2r //$ . ■

Para valores dados de  $n$  y  $b$ , deseamos elegir el valor de  $r$ , con  $r_b$ , que minimice la expresión  $.b=r/n C 2r /$ . Si  $b < blg n$ , entonces para cualquier valor de  $r_b$ , tenemos que  $.n C 2r / D .n/$ . Por lo tanto, elegir  $r = D b$  produce un tiempo de ejecución de  $.b=b/n C 2b/D .n/$ , que es asintóticamente óptimo. Si  $b \geq blg n$ , entonces elegir  $r = D blg n$  da el mejor tiempo dentro de un factor constante, que podemos ver a continuación. Elegir  $r = D blg n$  produce un tiempo de ejecución de  $.bn = lg n$ .

A medida que aumentamos  $r$  por encima de  $blg n$ , el término  $2r$  en el numerador aumenta más rápido que el término  $r$  en el denominador, por lo que aumentar  $r$  por encima de  $blg n$  produce un tiempo de ejecución de  $.bn = lg n$ . Si, en cambio, fuéramos a disminuir  $r$  por debajo de  $blg n$ , entonces el término  $b=r$  aumenta y el término  $n C 2r$  permanece en  $.n/$ .

¿Es preferible la clasificación radix a un algoritmo de clasificación basado en comparación, como la clasificación rápida? Si  $b = D O.lg n/$ , como suele ser el caso, y elegimos  $r = lg n$ , entonces el tiempo de ejecución de radix sort es  $.n/$ , que parece ser mejor que el tiempo de ejecución esperado de quicksort de  $.n lg n$ . Sin embargo, los factores constantes ocultos en la notación , difieren.

Aunque la ordenación radix puede realizar menos pasadas que la ordenación rápida sobre las  $n$  claves, cada pasada de la ordenación radix puede llevar mucho más tiempo. El algoritmo de ordenación que preferimos depende de las características de las implementaciones, de la máquina subyacente (p. ej., la ordenación rápida a menudo usa cachés de hardware con mayor eficacia que la ordenación radix) y de los datos de entrada. Además, la versión de ordenación radix que usa la ordenación por conteo como la ordenación estable intermedia no ordena en el lugar, lo que sí hacen muchas de las ordenaciones de comparación  $.n lg n$ -time. Por lo tanto, cuando el almacenamiento de la memoria principal es escaso, es posible que prefiramos un algoritmo en el lugar como Quicksort.

**Ejercicios****8.3-1**

Usando la Figura 8.3 como modelo, ilustre la operación de RADIX-SORT en la siguiente lista de palabras en inglés: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, GRANDE, TÉ, AHORA, ZORRO.

8.3-2

¿Cuáles de los siguientes algoritmos de clasificación son estables: clasificación por inserción, clasificación por fusión, clasificación en montón y clasificación rápida? Proporcione un esquema simple que haga estable cualquier algoritmo de clasificación. ¿Cuánto tiempo y espacio adicional implica su esquema?

8.3-3

Use la inducción para probar que la ordenación radix funciona. ¿Dónde necesita su prueba la suposición de que el tipo intermedio es estable?

8.3-4

Muestre cómo ordenar  $n$  enteros en el rango de 0 a  $n^3 - 1$  en  $O(n \log n)$  tiempo.

8.3-5 ?

En el primer algoritmo de clasificación de tarjetas de esta sección, ¿exactamente cuántas pasadas de clasificación se necesitan para clasificar números decimales de  $d$  dígitos en el peor de los casos? ¿De cuántos montones de cartas necesitaría hacer un seguimiento un operador en el peor de los casos?

---

#### 8.4 Clasificación de cubos

La ordenación de cubo asume que la entrada se extrae de una distribución uniforme y tiene un tiempo de ejecución de caso promedio de  $O(n \log n)$ . Al igual que la ordenación por conteo, la ordenación por cubos es rápida porque asume algo sobre la entrada. Mientras que la ordenación por conteo asume que la entrada consta de números enteros en un rango pequeño, la ordenación por cubo asume que la entrada se genera mediante un proceso aleatorio que distribuye los elementos de manera uniforme e independiente en el intervalo  $[0; 1]$ . (Consulte la Sección C.2 para obtener una definición de distribución uniforme).

La ordenación de cubeta divide el intervalo  $[0; 1]$  en  $n$  subintervalos de igual tamaño, o cubos, y luego distribuye los  $n$  números de entrada en los cubos. Dado que las entradas se distribuyen de manera uniforme e independiente sobre  $[0; 1]$ , no esperamos que caigan muchos números en cada cubeta. Para producir el resultado, simplemente clasificamos los números en cada cubo y luego revisamos los cubos en orden, enumerando los elementos en cada uno.

Nuestro código para la ordenación de cubos asume que la entrada es un arreglo  $A$  de  $n$  elementos y que cada elemento  $A[i]$  en el arreglo satisface  $0 \leq A[i] < 1$ . El código requiere un arreglo auxiliar  $B[0 : n-1]$  de listas enlazadas (cubos) y asume que existe un mecanismo para mantener dichas listas. (La Sección 10.2 describe cómo implementar operaciones básicas en listas enlazadas).

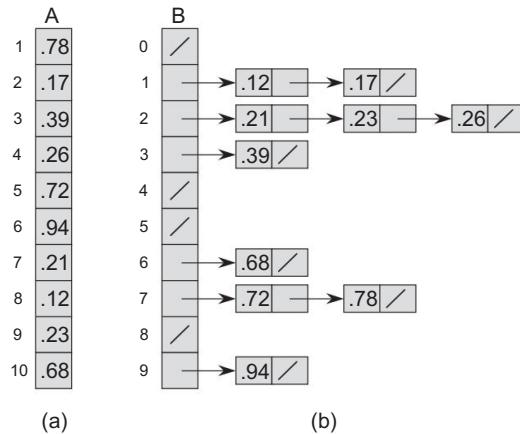


Figura 8.4 La operación de BUCKET-SORT para  $n = 10$ . (a) El arreglo de entrada  $A[1 : 10]$ . (b) El arreglo  $B[0 : 9]$  de 9 de listas ordenadas (cubos) después de la línea 8 del algoritmo. La cubeta  $i$  contiene valores en el intervalo semiabierto  $C[i-1 : i]$ . La salida ordenada consiste en una concatenación en el orden de las listas  $B[0 : 9]; B[1 : 10]; \dots; B[9 : 10]$ .

BUCKET-SORT.A/ 1 sea

BCE0 : : n 1 un nuevo arreglo 2 n D A:longitud

3 para i D 0 a n 1

4                hacer BCEi una lista vacía 5

para i D 1 a n

                  insertar ACEi en la lista BCEbnACEic 6

7 para i D 0 a n 1 ordenar la

                  lista BCEi con inserción ordenar 8 9

concatenar las listas BCE0; BCE1; : : ; BCEn 1 juntos en orden

La figura 8.4 muestra la operación de ordenación por cubeta en una matriz de entrada de 10 números.

Para ver que este algoritmo funciona, considere dos elementos  $A[i]$  y  $A[j]$ . Suponga sin pérdida de generalidad que  $A[i] > A[j]$ . Dado que  $b[A[i]] > b[A[j]]$ , el elemento  $A[i]$  va al mismo cubo que  $A[j]$  o va a un cubo con un índice más bajo. Si  $A[i]$  y  $A[j]$  van en el mismo cubo, entonces el ciclo for de las líneas 7 y 8 los coloca en el orden correcto. Si  $A[i]$  y  $A[j]$  van en cubos diferentes, entonces la línea 9 los pone en el orden correcto. Por lo tanto, la ordenación de depósitos funciona correctamente.

Para analizar el tiempo de ejecución, observe que todas las líneas, excepto la línea 8, toman tiempo de encendido/ en el peor de los casos. Necesitamos analizar el tiempo total que tardan las  $n$  llamadas en ordenar por inserción en la línea 8.

Para analizar el costo de las llamadas al ordenamiento por inserción, sea ni la variable aleatoria que denota el número de elementos colocados en el cubo  $BCE_i$ . Dado que la ordenación por inserción se ejecuta en tiempo cuadrático (consulte la Sección 2.2), el tiempo de ejecución de la ordenación por depósito es

$$T \cdot n / D \cdot n / CXn_1 \quad O.n2_i / : \\ iD0$$

Ahora analizamos el tiempo de ejecución del caso promedio del tipo de cubo, calculando el valor esperado del tiempo de ejecución, donde tomamos la expectativa sobre la distribución de entrada. Tomando las expectativas de ambos lados y usando la linealidad de la expectativa, tenemos

$$E \cdot ET \cdot n / DE " \cdot n / CXn_1 D \cdot n / CXn_1 / \# \\ E O.n2 / i \quad (\text{por linealidad de la expectativa}) \\ iD0 \\ D \cdot n / CXn_1 \quad O.E n2 / i \quad (\text{por ecuación (C.22)) .}) \quad (8.1)$$

Afirmamos que

$$E n2_i = 2 \cdot 1 = n \quad (8.2)$$

para  $i \in D[0; 1; \dots; n - 1]$ . No sorprende que cada cubeta  $i$  tenga el mismo valor de  $E \cdot En2$  ya que cada valor en el arreglo de entrada A tiene la misma probabilidad de caer en cualquier cubeta. Para probar la ecuación (8.2), definimos variables aleatorias indicadoras

$X_{ij} \in \{0, 1\}$  cae en balde si

para  $i \in D[0; 1; \dots; n - 1]$  y  $j \in D[1; 2; \dots; n]$ . De este modo,

$$n_i D X_n \quad X_{ij} : \\ jD1$$

Para calcular  $E \cdot En2$ , expandimos el cuadrado y reagrupamos términos:

$$E n_2$$

$$DE " Xn_{jD1} Xij!2#$$

$$DE " Xn_{jkD1} XijXik#$$

$$DE 2 \begin{matrix} xn & X2 \\ jD1 & 4 \end{matrix} CX \begin{matrix} & X \\ & 1j \text{ n } 1kn \text{ k} \neq j \end{matrix} xijxik \begin{matrix} 3 \\ 5 \end{matrix}$$

$$Xn \begin{matrix} E X2_{ij} \\ jD1 \end{matrix} CX \begin{matrix} & X \\ & 1j \text{ norte} \end{matrix} E \sigma XijXik ; \quad (8.3)$$

donde la última línea sigue por la linealidad de la expectativa. Evaluamos las dos sumas por separado. La variable aleatoria indicadora  $X_{ij}$  es 1 con probabilidad  $1=n$  y 0 en caso contrario, y por lo tanto

$$E X2_{ij} = D 12 \begin{matrix} 1 \\ \text{norte} \end{matrix} c_{02} \begin{matrix} - \\ 1 \\ \text{norte} \end{matrix} 1 \begin{matrix} 1 \\ \text{norte} \end{matrix}$$

$$D \begin{matrix} 1 \\ \text{norte} \end{matrix} ;$$

Cuando  $k \neq j$ , las variables  $X_{ij}$  y  $X_{ik}$  son independientes y, por lo tanto,

$$E \sigma XijXik = E Xij E Xik 1$$

$$\begin{matrix} 1 \\ n \\ \text{norte} \end{matrix} \begin{matrix} - \\ 1 \\ \text{norte} \end{matrix}$$

$$D \begin{matrix} 1 \\ \text{n}^2 \end{matrix} ;$$

Sustituyendo estos dos valores esperados en la ecuación (8.3), obtenemos

$$E n_2 \begin{matrix} Xn \\ jD1 \end{matrix} \begin{matrix} 1 \\ \text{norte} \end{matrix} CX \begin{matrix} & X \\ & 1j \text{ n } 1kn \text{ k} \neq j \end{matrix} \begin{matrix} 1 \\ n^2 \end{matrix}$$

$$Dn \begin{matrix} 1 \\ \text{norte} \end{matrix} Cnn \begin{matrix} 1/ \\ n^2 \end{matrix}$$

$$D 1 C \begin{matrix} n \\ 1 \\ \text{norte} \end{matrix}$$

$$D 2 \begin{matrix} 1 \\ \text{norte} \end{matrix} ;$$

lo que prueba la ecuación (8.2).

Usando este valor esperado en la ecuación (8.1), concluimos que el caso promedio el tiempo de ejecución para la ordenación de depósitos es  $,n/ C n O.2 1=n/ D ,n/$ .

Incluso si la entrada no se extrae de una distribución uniforme, la ordenación de depósitos aún puede ejecutarse en tiempo lineal. Siempre que la entrada tenga la propiedad de que la suma de los cuadrados de los tamaños de los cubos es lineal en el número total de elementos, la ecuación (8.1) nos dice que la ordenación de los cubos se ejecutará en un tiempo lineal.

### Ejercicios

#### 8.4-1

Usando la Figura 8.4 como modelo, ilustre la operación de BUCKET-SORT en el arreglo AD h:79; :13; :dieciséis; :64; :39; :20; :89; :53; :71; :42i.

#### 8.4-2

Explique por qué el tiempo de ejecución en el peor de los casos para la ordenación de depósitos es  $,n^2/$ . ¿Qué simple cambio en el algoritmo preserva su tiempo de ejecución de caso promedio lineal y hace que su tiempo de ejecución en el peor de los casos sea  $O(n \lg n)$ ?

#### 8.4-3

Sea  $X$  una variable aleatoria que es igual al número de caras en dos lanzamientos de una moneda justa. ¿Qué es  $E(X)$ ? ¿Qué es  $E^2(X)$ ?

#### 8.4-4 ?

Nos dan  $n$  puntos en el círculo unitario,  $p_i(x_i, y_i)$ , tal que  $0 < x_i^2 + y_i^2 \leq 1$ . Supongamos que los puntos están uniformemente distribuidos; es decir, la probabilidad de encontrar un punto en cualquier región del círculo es proporcional al área de esa región.

Diseñe un algoritmo con un tiempo de ejecución de caso promedio de  $,n/$  para ordenar los  $n$  puntos por sus distancias  $d_i$  desde el origen. (Sugerencia: diseñe los tamaños de cubo en BUCKET-SORT para reflejar la distribución uniforme de los puntos en el círculo unitario).

#### 8.4-5 ?

Una función de distribución de probabilidad  $P(x)$  para una variable aleatoria  $X$  está definida por  $P(x) = \Pr(X \leq x)$ . Supongamos que dibujamos una lista de  $n$  variables aleatorias  $X_1, X_2, \dots, X_n$  de una función de distribución de probabilidad continua  $P$  que es computable en  $O(1/t)$  tiempo. Proporcione un algoritmo que clasifique estos números en tiempo de caso promedio lineal.

---

## Problemas

### 8-1 Límites inferiores probabilísticos en la clasificación por comparación

En este problema, demostramos un límite inferior probabilístico  $n \lg n$  en el tiempo de ejecución de cualquier clasificación por comparación determinista o aleatoria en  $n$  elementos de entrada distintos. Comenzamos examinando una comparación determinista tipo A con un árbol de decisión TA. Suponemos que cada permutación de las entradas de A es igualmente probable.

- a. Suponga que cada hoja de TA está etiquetada con la probabilidad de que se alcance dada una entrada aleatoria. Demuestre que exactamente  $n$  hojas están etiquetadas con  $1=n$  y que el resto están etiquetadas con 0.
- b. Sea DT / la longitud del camino externo de un árbol de decisión T ; es decir, DT/ es la suma de las profundidades de todas las hojas de T. Sea T un árbol de decisión con  $k>1$  hojas, y sean LT y RT los subárboles izquierdo y derecho de T. Demuestre que  $DT / D D.LT/ C D.RT/ C k$ .
- C. Sea dk/ el valor mínimo de DT/ sobre todos los árboles de decisión T con  $k>1$  hojas. Demuestre que  $dk/ D \min_{1 \leq i \leq k} fd.i / C dk i / C kg$ . (Sugerencia: considere un árbol de decisión T con  $k$  hojas que alcanza el mínimo. Sea  $i_0$  el número de hojas en LT y  $k - i_0$  el número de hojas en RT).
- d. Demuestre que para un valor dado de  $k>1$  e i en el rango  $1 \leq i \leq k$ , la función  $i \lg i C .k_i / \lg k_i$  se minimiza en  $i = k/2$ . Concluya que  $dk/ D .k \lg k/$ .

mi. Demuestre que  $D.TA/ D .n \lg n//$ , y concluya que el tiempo promedio del caso para ordenar  $n$  elementos es  $n \lg n$ .

Ahora, considere una comparación aleatoria tipo B. Podemos extender el modelo de árbol de decisión para manejar la aleatorización mediante la incorporación de dos tipos de nodos: nodos de comparación ordinarios y nodos de "aleatorización". Un nodo de aleatorización modela una elección aleatoria de la forma RANDOM.1; r/ hecha por el algoritmo B; el nodo tiene  $r$  hijos, cada uno de los cuales tiene la misma probabilidad de ser elegido durante la ejecución del algoritmo.

F. Muestre que para cualquier tipo de comparación aleatoria B, existe un tipo de comparación determinista A cuyo número esperado de comparaciones no es mayor que las realizadas por B.

### 8-2 Clasificación en el lugar en tiempo lineal

Suponga que tenemos una matriz de  $n$  registros de datos para clasificar y que la clave de cada registro tiene el valor 0 o 1. Un algoritmo para clasificar tal conjunto de registros podría poseer algún subconjunto de los siguientes tres características deseables:

1. El algoritmo se ejecuta en  $O(n)$  time.
  2. El algoritmo es estable.
  3. El algoritmo ordena en el lugar, usando no más que una cantidad constante de espacio de almacenamiento además del arreglo original.
    - a. Proporcione un algoritmo que satisfaga los criterios 1 y 2 anteriores.
    - b. Proporcione un algoritmo que satisfaga los criterios 1 y 3 anteriores.
- C. Dé un algoritmo que satisfaga los criterios 2 y 3 anteriores.
- d. ¿Puede usar alguno de sus algoritmos de clasificación de las partes (a)–(c) como el método de clasificación utilizado en la línea 2 de RADIX-SORT, para que RADIX-SORT clasifique  $n$  registros con claves de  $b$  bits en  $O(bn)$  tiempo? Explique cómo o por qué no.
- mi. Suponga que los  $n$  registros tienen claves en el rango de 1 a  $k$ . Muestre cómo modificar la clasificación de conteo para que clasifique los registros en lugar en  $O(kn)$  time. Puede usar  $O(k)$  almacenamiento fuera de la matriz de entrada. ¿Es estable su algoritmo? (Pista: ¿Cómo lo harías para  $k = 3$ ?)

### 8-3 Clasificación de elementos de longitud

variable a. Se le da una matriz de enteros, donde diferentes enteros pueden tener diferentes números de dígitos, pero el número total de dígitos sobre todos los enteros en la matriz es  $n$ . Muestre cómo ordenar la matriz en  $O(n)$  time.

- b. Se le proporciona una matriz de cadenas, donde diferentes cadenas pueden tener diferentes cantidades de caracteres, pero la cantidad total de caracteres en todas las cadenas es  $n$ . Muestre cómo ordenar las cadenas en  $O(n)$  time.

(Tenga en cuenta que el orden deseado aquí es el orden alfabético estándar; por ejemplo,  $a < ab < b$ ).

### 8-4 Jarras de agua

Supón que te dan  $n$  jarras de agua rojas y  $n$  azules, todas de diferentes formas y tamaños. Todas las jarras rojas contienen diferentes cantidades de agua, al igual que las azules. Además, por cada jarra roja, hay una jarra azul que contiene la misma cantidad de agua, y viceversa.

Tu tarea es encontrar una agrupación de las jarras en pares de jarras rojas y azules que contengan la misma cantidad de agua. Para ello, puedes realizar la siguiente operación: elige un par de jarras en las que una sea roja y la otra azul, llena la jarra roja con agua y luego vierte el agua en la jarra azul. Esta operación te dirá si la jarra roja o la azul pueden contener más agua, o si tienen el mismo volumen. Suponga que tal comparación toma una unidad de tiempo. Su objetivo es encontrar un algoritmo que haga un número mínimo de comparaciones para determinar la agrupación. Recuerda que no puedes comparar directamente dos jarras rojas o dos jarras azules.

a. Describa un algoritmo determinista que use comparaciones  $\leq n^2$  para agrupar los

jarras en pares.

b. Demuestre un límite inferior de  $n \lg n$  para el número de comparaciones que debe hacer un algoritmo que resuelva este problema.

C. Proporcione un algoritmo aleatorio cuyo número esperado de comparaciones sea  $O(n \lg n)$  y demuestre que este límite es correcto. ¿Cuál es el número de comparaciones en el peor de los casos para su algoritmo?

#### 8-5 Ordenación promedio

Suponga que, en lugar de ordenar una matriz, solo requerimos que los elementos aumenten en promedio.

Más precisamente, llamamos a un arreglo de  $n$  elementos  $A$   $k$ -ordenado si, para todo  $i < k$ , se cumple lo siguiente:  $A[i] \leq A[i+1] \leq \dots \leq A[k]$ . Elegir

$$\frac{\text{elegir } A[1]}{k} \quad \frac{\text{elegir } A[2]}{k} \quad \dots \quad \frac{\text{elegir } A[k]}{k}$$

a. ¿Qué significa que una matriz esté ordenada por 1?

b. Dar una permutación de los números  $1, 2, \dots, 10$  que está clasificado en 2, pero no clasificado.

C. Demuestre que un arreglo de  $n$  elementos está ordenado en  $k$  si y solo si  $A[i] \leq A[i+k]$  para todos  $i = 0, 1, \dots, n-k-1$ .

d. Proporcione un algoritmo que clasifique con  $k$  una matriz de  $n$  elementos en  $O(n \lg n + k)$  tiempo.

También podemos mostrar un límite inferior en el tiempo para producir una matriz ordenada  $k$ , cuando  $k$  es una constante

mi. Muestre que podemos ordenar una matriz ordenada por  $k$  de longitud  $n$  en  $O(n \lg k)$  tiempo. (Sugerencia: utilice la solución del ejercicio 6.5-9).

F. Muestre que cuando  $k$  es una constante,  $k$ -ordenar un arreglo de  $n$  elementos requiere  $n \lg n$  de tiempo. (Sugerencia: use la solución de la parte anterior junto con el límite inferior en ordenaciones de comparación).

## 8-6 Límite inferior en la fusión de listas ordenadas El

problema de fusionar dos listas ordenadas surge con frecuencia. Hemos visto un procedimiento para ello como la subrutina MERGE en la Sección 2.3.1. En este problema, demostraremos un límite inferior de  $2n - 1$  en el peor de los casos, el número de comparaciones requeridas para fusionar dos listas ordenadas, cada una con  $n$  elementos.

Primero mostraremos un límite inferior de  $2n - 1$  comparaciones usando una decisión árbol.

a. Dados  $2n$  números, calcule el número de formas posibles de dividirlos en dos listas ordenadas, cada una con  $n$  números.

b. Usando un árbol de decisión y su respuesta a la parte (a), demuestre que cualquier algoritmo que combine correctamente dos listas ordenadas debe realizar al menos  $2n - 1$  comparaciones.

Ahora mostraremos un límite  $2n - 1$  ligeramente más estrecho.

C. Muestre que si dos elementos son consecutivos en el orden ordenado y de diferentes listas, entonces deben compararse.

d. Use su respuesta a la parte anterior para mostrar un límite inferior de comparaciones  $2n - 1$  para fusionar dos listas ordenadas.

## 8-7 El lema de clasificación 0-1 y clasificación por columnas

Una operación de comparación e intercambio en dos elementos de matriz  $A[i:j]$ , donde  $i < j$ , tiene la forma

COMPARAR-CAMBIO.A; i; j / 1 si  $A[i] >$

$A[j]$  2 intercambia

$A[i]$  con  $A[j]$

Después de la operación de comparación e intercambio, sabemos que  $A[i] \leq A[j]$ .

Un algoritmo de intercambio de comparación ajeno opera únicamente mediante una secuencia de operaciones de intercambio de comparación especificadas previamente. Los índices de las posiciones comparadas en la secuencia deben determinarse de antemano, y aunque pueden depender del número de elementos que se ordenan, no pueden depender de los valores que se ordenan, ni pueden depender del resultado de cualquier intercambio de comparación anterior. operación.

Por ejemplo, aquí está la ordenación por inserción expresada como un algoritmo de comparación e intercambio ajeno:

INSERCIÓN-

CLASIFICACIÓN.A/ 1 para  $j \leq D[2]$

$\wedge A[:longitud 2 para  $i \leq D[j - 1]$  hasta 1]$

3

COMPARAR-INTERCAMBIO.A; i; yo C 1/

El lema de clasificación 0-1 proporciona una forma poderosa de demostrar que un algoritmo de comparación e intercambio inconsciente produce un resultado ordenado. Establece que si un algoritmo de comparación e intercambio ajeno clasifica correctamente todas las secuencias de entrada que consisten solo en 0 y 1, entonces clasifica correctamente todas las entradas que contienen valores arbitrarios.

Demostrará el lema de ordenación 0-1 demostrando su contraposición: si un algoritmo de comparación e intercambio ajeno falla al ordenar una entrada que contiene valores arbitrarios, entonces falla al ordenar alguna entrada 0-1. Suponga que un algoritmo de comparación-intercambio olvidado X no ordena correctamente el arreglo  $A \in \{0, 1\}^n$ . Sea  $A_{\text{Eq}}$  el valor más pequeño en  $A$  que el algoritmo X coloca en la ubicación incorrecta, y sea  $A_{\text{Eq}}$  el valor que el algoritmo X mueve a la ubicación a la que debería haber ido  $A_{\text{Eq}}$ . Defina una matriz  $B \in \{0, 1\}^n$  de la siguiente manera:

$$B_{ij} = \begin{cases} 1 & \text{si } A_{ij} > A_{\text{Eq}} \\ 0 & \text{si } A_{ij} \leq A_{\text{Eq}} \end{cases}$$

- a. Argumente que  $A_{\text{Eq}} > A_{\text{Eq}}$ , de modo que  $B_{\text{Eq}} = 0$  y  $B_{\text{Eq}} = 1$ .
- b. Para completar la prueba del lema de ordenación 0-1, demuestre que el algoritmo X no ordena la matriz B correctamente.

Ahora utilizará el lema de clasificación 0-1 para demostrar que un algoritmo de clasificación en particular funciona correctamente. El algoritmo, columnsort, funciona en una matriz rectangular de  $n$  elementos. La matriz tiene  $r$  filas y  $s$  columnas (de modo que  $n = r \cdot s$ ), sujeto a tres restricciones:

$r$  debe ser par,

$s$  debe ser un divisor de  $r$ , y

$r \geq 2s^2$ .

Cuando se completa columnsort, la matriz se ordena en orden de columna principal: al leer las columnas, de izquierda a derecha, los elementos aumentan monótonamente.

Columnsort opera en ocho pasos, independientemente del valor de  $n$ . Los pasos impares son todos iguales: ordenar cada columna individualmente. Cada paso par es una permutación fija. Aquí están los pasos:

1. Ordene cada columna.
2. Transponga la matriz, pero vuelva a darle forma a  $r$  filas y  $s$  columnas. En otras palabras, convierta la columna más a la izquierda en las filas superiores  $r=s$ , en orden; convierta la siguiente columna en las siguientes filas  $r=s$ , en orden; etcétera.
3. Ordene cada columna.
4. Realice el inverso de la permutación realizada en el paso 2.

10 14 5	412	4 8 10	136	1 4 11
8 7 17	835	12 16 18	257	3 8 14
12 1 6	10 7 6	137	4 8 10	6 10 17
16 9 11	12 9 11	9 14 15	9 13 15	2 9 12
4 15 2	16 14 13	256	11 14 17	5 13 16
18 3 13	18 15 17	11 13 17	12 16 18	7 15 18

(a)

(b)

(C)

(d)

(mi)

1 4 11 2 8	5 10 16 6	4 10 16 5	1 7 13
12	13 17	11 17	2 8 14
3 9 14 5 10	7 15 18 1	6 12 18 1	3 9 15 4 10
16 6 13 17	4 11 2 8	7 13 2 8	16 5 11 17
7 15 18	12 3 9 14	14 3 9 15	6 12 18

(F)

(gramo)

(h)

(i)

Figura 8.5 Los pasos del ordenamiento por columnas. (a) La matriz de entrada con 6 filas y 3 columnas. (b) Después de ordenar cada columna en el paso 1. (c) Después de transponer y remodelar en el paso 2. (d) Después de ordenar cada columna en el paso 3. (e) Después de realizar el paso 4, que invierte la permutación del paso 2. (f) Después de ordenar cada columna en el paso 5. (g) Después de cambiar media columna en el paso 6. (h) Después de ordenar cada columna en el paso 7. (i) Después de realizar el paso 8, que invierte la permutación del paso 6. La matriz ahora está ordenada en orden de columna principal.

5. Ordene cada columna.

6. Desplace la mitad superior de cada columna a la mitad inferior de la misma columna y desplace la mitad inferior de cada columna a la mitad superior de la siguiente columna a la derecha. Deje vacía la mitad superior de la columna más a la izquierda. Cambie la mitad inferior de la última columna a la mitad superior de una nueva columna más a la derecha y deje vacía la mitad inferior de esta nueva columna.

7. Ordene cada columna.

8. Realice el inverso de la permutación realizada en el paso 6.

La figura 8.5 muestra un ejemplo de los pasos de clasificación por columnas con  $r = 6$  y  $s = 3$ .  $2s=2$ , sucede viola el requisito de que  $r$  funcione. (Aunque este ejemplo

C. Argumente que podemos tratar la clasificación por columnas como un algoritmo de comparación e intercambio inconsciente, incluso si no sabemos qué método de clasificación utilizan los pasos impares.

Aunque puede parecer difícil de creer que la ordenación por columnas realmente ordena, usará el lema de ordenación 0-1 para demostrar que así es. El lema de clasificación 0-1 se aplica porque podemos tratar la clasificación por columnas como un algoritmo de comparación e intercambio ajeno. A

Un par de definiciones lo ayudarán a aplicar el lema de clasificación 0-1. Decimos que un área de una matriz está limpia si sabemos que contiene solo ceros o solo unos. De lo contrario, el área puede contener ceros y unos mezclados y está sucia. De aquí en adelante, suponga que la matriz de entrada contiene solo 0 y 1, y que podemos tratarla como una matriz con  $r$  filas y  $s$  columnas.

d. Demuestre que después de los pasos 1 a 3, el arreglo consta de algunas filas limpias de 0 en la parte superior, algunas filas limpias de 1 en la parte inferior y, como máximo, filas sucias entre ellas.

mi. Demuestre que después del paso 4, el arreglo, leído en orden de columna principal, comienza con un área limpia de 0, termina con un área limpia de 1 y tiene un área sucia de como máximo  $s_2$  elementos en el medio.

F. Demuestre que los pasos 5 a 8 producen una salida 0-1 completamente ordenada. Concluya que la clasificación por columnas clasifica correctamente todas las entradas que contienen valores arbitrarios.

gramo. Ahora suponga que  $s$  no divide a  $r$ . Demuestre que después de los pasos 1 a 3, la matriz consta de algunas filas limpias de 0 en la parte superior, algunas filas limpias de 1 en la parte inferior y, como máximo, 2 filas 1 sucias entre ellas. ¿Qué tan grande debe ser  $r$ , en comparación con  $s$ , para que columnsort ordene correctamente cuando  $s$  no divide a  $r$ ?

H. Sugiera un cambio simple al paso 1 que nos permita mantener el requisito de que  $r \geq s^2$  incluso cuando  $s$  no divide a  $r$ , y demuestre que con su cambio, columnsort ordena correctamente.

## Notas del capítulo

Ford y Johnson [110] introdujeron el modelo de árbol de decisión para estudiar los géneros de comparación. El extenso tratado de Knuth sobre la clasificación [211] cubre muchas variaciones del problema de la clasificación, incluido el límite inferior teórico de la información sobre la complejidad de la clasificación que se presenta aquí. Ben-Or [39] estudió los límites inferiores para la clasificación utilizando generalizaciones del modelo de árbol de decisión.

Knuth atribuye a HH Seward la invención de la ordenación por conteo en 1954, así como la idea de combinar la ordenación por conteo con la ordenación por radix. La clasificación Radix que comienza con el dígito menos significativo parece ser un algoritmo popular ampliamente utilizado por los operadores de máquinas clasificadoras de tarjetas mecánicas. Según Knuth, la primera referencia publicada al método es un documento de 1929 de LJ Comrie que describe el equipo de tarjetas perforadas. La clasificación por cubos se ha utilizado desde 1956, cuando EJ Isaac y RC Singleton [188] propusieron la idea básica.

Munro y Raman [263] dan un algoritmo de clasificación estable que realiza comparaciones  $O(nC)$  en el peor de los casos, donde  $0 < C \leq 1$  es cualquier constante fija. A pesar de

cualquiera de los algoritmos  $O(n \lg n)$ -time hace menos comparaciones, el algoritmo de Munro y Raman mueve datos solo  $O(n)$  times y opera en su lugar.

Muchos investigadores han considerado el caso de clasificar  $n$  enteros de  $b$  bits en un tiempo  $\lg n$ . Se han obtenido varios resultados positivos, cada uno bajo supuestos ligeramente diferentes sobre el modelo de cálculo y las restricciones impuestas al algoritmo. Todos los resultados asumen que la memoria de la computadora está dividida en palabras  $b$ -bit direccionables. Fredman y Willard [115] introdujeron la estructura de datos del árbol de fusión y la usaron para ordenar  $n$  enteros en  $O(n \lg n = \lg \lg n)$  time. Este límite fue posteriormente mejorado a  $O(n \lg \lg n)$  time por Andersson [16]. Estos algoritmos requieren el uso de la multiplicación y varias constantes precalculadas. Andersson, Hagerup, Nilsson y Raman [17] han mostrado cómo ordenar  $n$  enteros en  $O(n \lg \lg n)$  time sin usar la multiplicación, pero su método requiere un almacenamiento que puede ser ilimitado en términos de  $n$ . Mediante el uso de hashing multiplicativo, podemos reducir el almacenamiento necesario a  $O(n)$ , pero luego el límite en el peor de los casos  $O(n \lg \lg n)$  en el tiempo de ejecución se convierte en un límite de tiempo esperado. Generalizando los árboles de búsqueda exponencial de Andersson [16], Thorup [335] proporcionó un algoritmo de clasificación  $O(n \lg n^{1/2})$ -time que no usa multiplicación o aleatorización, y usa espacio lineal. Combinando estas técnicas con algunas ideas nuevas, Han [158] mejoró el límite para ordenar a  $O(n \lg n \lg \lg n)$  time. Aunque estos algoritmos son avances teóricos importantes, todos son bastante complicados y, en la actualidad, parece poco probable que compitan con los algoritmos de clasificación existentes en la práctica.

El algoritmo de clasificación por columnas del problema 8-7 es de Leighton [227].

---

## 9 Estadísticas de medianas y pedidos

El estadístico de  $i$ -ésimo orden de un conjunto de  $n$  elementos es el  $i$ -ésimo elemento más pequeño. Por ejemplo, el mínimo de un conjunto de elementos es el estadístico de primer orden ( $i = 1$ ), y el máximo es el estadístico de orden  $n$  ( $i = n$ ). Una mediana, informalmente, es el "punto medio" del conjunto. Cuando  $n$  es impar, la mediana es única y ocurre en  $i = \lfloor n/2 \rfloor + 1$ . Cuando  $n$  es par, hay dos medianas, que ocurren en  $i = \lfloor n/2 \rfloor$  e  $i = \lfloor n/2 \rfloor + 1$ . Por lo tanto, independientemente de la paridad de  $n$ , las medianas ocurren en  $i = \lfloor n/2 \rfloor + 1$  (la mediana inferior) y  $i = \lfloor n/2 \rfloor + 2$  (la mediana superior). Sin embargo, para simplificar este texto, usamos constantemente la frase "la mediana" para referirnos a la mediana inferior.

Este capítulo aborda el problema de seleccionar el estadístico de  $i$ -ésimo orden de un conjunto de  $n$  números distintos. Suponemos por conveniencia que el conjunto contiene números distintos, aunque virtualmente todo lo que hacemos se extiende a la situación en la que un conjunto contiene valores repetidos. Especificamos formalmente el problema de selección de la siguiente manera:

Entrada: Un conjunto  $A$  de  $n$  números (distintos) y un entero  $i$ , con  $1 \leq i \leq n$ .

Salida: El elemento  $x \in A$  que es más grande que exactamente  $i - 1$  otros elementos de  $A$ .

Podemos resolver el problema de selección en  $O(n \lg n)$  time, ya que podemos ordenar los números usando heapsort o merge sort y luego simplemente indexar el  $i$ -ésimo elemento en la matriz de salida. Este capítulo presenta algoritmos más rápidos.

En la Sección 9.1, examinamos el problema de seleccionar el mínimo y el máximo de un conjunto de elementos. Más interesante es el problema de selección general, que investigamos en las dos secciones siguientes. La Sección 9.2 analiza un algoritmo aleatorizado práctico que logra un tiempo de ejecución  $O(n)$  esperado, suponiendo elementos distintos. La sección 9.3 contiene un algoritmo de interés más teórico que logra el tiempo de encendido/ejecución en el peor de los casos.

## 9.1 Mínimo y máximo

¿Cuántas comparaciones son necesarias para determinar el mínimo de un conjunto de  $n$  elementos? Podemos obtener fácilmente un límite superior de  $n - 1$  comparaciones: examinar cada elemento del conjunto por turnos y realizar un seguimiento del elemento más pequeño visto hasta el momento. En el siguiente procedimiento, asumimos que el conjunto reside en el arreglo A, donde A:longitud D n.

```
MINIMO.A/ 1
min D A[1] 2
para i D 2 a A:longitud 3 si
    min > A[i] min D A[i]
5 retorno min
```

Por supuesto, también podemos encontrar el máximo con  $n - 1$  comparaciones.

Es esto lo mejor que podemos hacer? Sí, ya que podemos obtener una cota inferior de  $n - 1$  comparaciones para el problema de determinación del mínimo. Piense en cualquier algoritmo que determine el mínimo como un torneo entre los elementos. Cada comparación es un partido en el torneo en el que gana el menor de los dos elementos.

Observando que todos los elementos excepto el ganador deben perder al menos un partido, concluimos que son necesarias  $n - 1$  comparaciones para determinar el mínimo. Por lo tanto, el algoritmo MINIMO es óptimo con respecto al número de comparaciones realizadas.

### Mínimo y máximo simultáneos

En algunas aplicaciones, debemos encontrar tanto el mínimo como el máximo de un conjunto de  $n$  elementos. Por ejemplo, un programa de gráficos puede necesitar escalar un conjunto de  $x, y$  datos para caber en una pantalla de visualización rectangular u otro dispositivo de salida gráfica. Para hacerlo, el programa primero debe determinar el valor mínimo y máximo de cada coordenada.

En este punto, debería ser obvio cómo determinar tanto el mínimo como el máximo de  $n$  elementos usando comparaciones  $n$ , lo cual es asintóticamente óptimo: simplemente encuentre el mínimo y el máximo de forma independiente, usando  $n - 1$  comparaciones para cada uno, para un total de  $2n - 2$  comparaciones.

De hecho, podemos encontrar tanto el mínimo como el máximo utilizando como máximo  $3bn - 2c$  comparaciones. Lo hacemos manteniendo los elementos mínimos y máximos vistos hasta ahora. En lugar de procesar cada elemento de la entrada comparándolo con el mínimo y el máximo actuales, a un costo de 2 comparaciones por elemento,

procesamos elementos en pares. Primero comparamos pares de elementos de la entrada entre sí, y luego comparamos el más pequeño con el mínimo actual y el más grande con el máximo actual, a un costo de 3 comparaciones por cada 2 elementos.

La forma en que configuramos los valores iniciales para el mínimo y el máximo actual depende de si  $n$  es par o impar. Si  $n$  es impar, establecemos tanto el mínimo como el máximo en el valor del primer elemento, y luego procesamos el resto de los elementos en pares. Si  $n$  es par, realizamos 1 comparación en los primeros 2 elementos para determinar los valores iniciales del mínimo y máximo, y luego procesamos el resto de los elementos en pares como en el caso de  $n$  impar.

Analicemos el número total de comparaciones. Si  $n$  es impar, entonces realizamos  $3 \frac{n}{2} = 2c$  comparaciones. Si  $n$  es par, realizamos 1 comparación inicial seguida de  $3 \cdot \frac{n}{2} - 2$  comparaciones, para un total de  $3n/2 - 2$ . Por lo tanto, en cualquier caso, el número total de comparaciones es como máximo  $3 \frac{n}{2} = 2c$ .

### Ejercicios

#### 9.1-1

Muestre que el segundo más pequeño de  $n$  elementos se puede encontrar con  $n \lceil \lg n \rceil - 2$  comparaciones en el peor de los casos. (Sugerencia: encuentre también el elemento más pequeño).

#### 9.1-2 ?

Demuestre el límite inferior de  $\lceil \frac{3n}{2} - 2 \rceil$  comparaciones en el peor de los casos para encontrar tanto el máximo como el mínimo de  $n$  números. (Sugerencia: Considere cuántos números son potencialmente el máximo o el mínimo, e investigue cómo una comparación afecta estos conteos).

## 9.2 Selección en tiempo lineal esperado

El problema general de selección parece más difícil que el simple problema de encontrar un mínimo. Sin embargo, sorprendentemente, el tiempo de ejecución asintótico para ambos problemas es el mismo:  $\lceil \frac{n}{2} \rceil$ . En esta sección, presentamos un algoritmo divide y vencerás para el problema de selección. El algoritmo RANDOMIZED-SELECT está modelado según el algoritmo de clasificación rápida del Capítulo 7. Al igual que en la clasificación rápida, dividimos la matriz de entrada recursivamente. Pero a diferencia de quicksort, que procesa recursivamente ambos lados de la partición, RANDOMIZED-SELECT funciona solo en un lado de la partición. Esta diferencia aparece en el análisis: mientras que quicksort tiene un tiempo de ejecución esperado de  $\lceil \frac{n}{2} \lg n \rceil$ , el tiempo de ejecución esperado de RANDOMIZED-SELECT es  $\lceil \frac{n}{2} \rceil$ , asumiendo que los elementos son distintos.

SELECCIÓN ALEATORIA utiliza el procedimiento PARTICIÓN ALEATORIA presentado en la Sección 7.3. Así, como RANDOMIZED-QUICKSORT, es un algoritmo aleatorio, ya que su comportamiento está determinado en parte por la salida de un generador de números aleatorios. El siguiente código para RANDOMIZED-SELECT devuelve el  $i$ -ésimo elemento más pequeño de la matriz  $A \in \mathbb{R}^n$ :

```

ALEATORIO-SELECCIÓN.A; pag; r; i /
1 si p == r 2
devuelve  $A[1:r]$ 
PARTICIÓN ALEATORIA.A; pag; r/4 k D qp C 1 5
si  $i == k$  6 devuelve
 $A[7:k]$  // el valor pivote es la respuesta
contrario  $i < k$  8
devuelve
RANDOMIZED-SELECT.A; pag; q 1; i / 9 else devuelve
RANDOMIZED-SELECT.A; q C 1; r; yo/

```

El procedimiento de SELECCIÓN ALEATORIA funciona de la siguiente manera. La línea 1 comprueba el caso base de la recursividad, en el que el subarreglo  $A[1:r]$  consta de un solo elemento. En este caso,  $i$  debe ser igual a 1, y simplemente devolvemos  $A[1:r]$  en la línea 2 como el  $i$ -ésimo elemento más pequeño. De lo contrario, la llamada a RANDOMIZED-PARTITION en la línea 3 divide el arreglo  $A[1:r]$  en dos subarreglos (posiblemente vacíos)  $A[1:q]$  y  $A[q+1:r]$  tales que cada elemento de  $A[1:q]$  es menor que o igual a  $A[q]$ , que a su vez es menor que cada elemento de  $A[q+1:r]$ . Al igual que en la ordenación rápida, nos referiremos a  $A[q]$  como el elemento pivote. La línea 4 calcula el número  $k$  de elementos en el subarreglo  $A[1:q]$ , es decir, el número de elementos en el lado inferior de la partición, más uno para el elemento pivote. La línea 5 luego verifica si  $A[q]$  es el  $i$ -ésimo elemento más pequeño. Si es así, entonces la línea 6 devuelve  $A[q]$ . De lo contrario, el algoritmo determina en cuál de los dos subarreglos  $A[1:q]$  y  $A[q+1:r]$  se encuentra el  $i$ -ésimo elemento más pequeño. Si  $i < k$ , entonces el elemento deseado se encuentra en el lado inferior de la partición y la línea 8 lo selecciona recursivamente del subarreglo. Sin embargo, si  $i > k$ , entonces el elemento deseado se encuentra en el lado alto de la partición. Como ya conocemos  $k$  valores que son más pequeños que el  $i$ -ésimo elemento más pequeño de  $A[1:r]$ —es decir, los elementos de  $A[1:q]$ —el elemento deseado es el  $i-k$ -ésimo elemento más pequeño de  $A[q+1:r]$ , cuya línea 9 encuentra recursivamente. El código parece permitir llamadas recursivas a subarreglos con 0 elementos, pero el Ejercicio 9.2-1 le pide que demuestre que esta situación no puede ocurrir.

El tiempo de ejecución en el peor de los casos para RANDOMIZED-SELECT es  $\Theta(n^2)$ , incluso para encontrar el mínimo, porque podríamos tener mucha mala suerte y siempre dividir alrededor del elemento restante más grande, y la partición lleva  $\Theta(n)$  tiempo. Veremos eso

Sin embargo, el algoritmo tiene un tiempo de ejecución esperado lineal y, debido a que es aleatorio, ninguna entrada en particular provoca el peor de los casos.

Para analizar el tiempo de ejecución esperado de RANDOMIZED-SELECT, dejamos que el tiempo de ejecución en una matriz de entrada  $A \in \mathbb{R}^{n \times n}$  sea una variable aleatoria que denotamos por  $T(n)$ , y obtenemos un límite superior en  $E[T(n)]$  de la siguiente manera. Es igualmente probable que el procedimiento RANDOMIZED-PARTITION devuelva cualquier elemento como pivote. Por lo tanto, para cada  $k$  tal que  $1 \leq k \leq n$ , el subarreglo  $A[p : q]$  tiene  $k$  elementos (todos menores o iguales al pivote) con probabilidad  $1/n$ . Para  $k = 1, 2, \dots, n$ , definimos variables aleatorias indicadoras  $X_k$  donde

$X_k = 1$  si el subarreglo  $A[p : q]$  tiene exactamente  $k$  elementos; ;

y así, suponiendo que los elementos son distintos, tenemos

$$E[X_k] = \frac{1}{n} \quad (9.1)$$

Cuando llamamos a RANDOMIZED-SELECT y elegimos  $A[eq]$  como elemento pivote, no sabemos, a priori, si terminaremos inmediatamente con la respuesta correcta, si recurrimos en el subarreglo  $A[p : q]$ , o recurrimos en el subarreglo  $A[eq : r]$ .

Esta decisión depende de dónde cae el  $i$ -ésimo elemento más pequeño en relación con  $A[eq]$ . Suponiendo que  $T(n)$  aumenta monótonamente, podemos establecer un límite superior del tiempo necesario para la llamada recursiva por el tiempo necesario para la llamada recursiva en la entrada más grande posible. En otras palabras, para obtener un límite superior, asumimos que el  $i$ -ésimo elemento está siempre en el lado de la partición con mayor número de elementos. Para una llamada dada de RANDOMIZED-SELECT, la variable aleatoria indicadora  $X_k$  tiene el valor 1 para exactamente un valor de  $k$ , y es 0 para todos los demás  $k$ . Cuando  $X_k = 1$ , los dos subarreglos en los que podríamos recurrir tienen tamaños  $k$  y  $n-k$ . Por lo tanto, nosotras tener la recurrencia

$$T(n) = \sum_{k=1}^n X_k T(n-k) + C \quad (9.2)$$

$$X_k = \begin{cases} 1 & \text{si } A[eq] \text{ es el } k\text{-ésimo menor elemento} \\ 0 & \text{de lo contrario} \end{cases}$$

Tomando los valores esperados, tenemos

$$E \text{CET}_{.n}/$$

$$E " X_{D1} X_k T_{.max.k} 1; nk// C On/# D X_n$$

$$\frac{E \text{CET}_{.n}/}{kD1} = E \text{CET}_{.max.k} 1; nk// C encendido/ \quad (\text{por linealidad de la expectativa})$$

$$\frac{X_n}{kD1} = E \text{CET}_{.max.k} 1; nk// C On/ \quad (\text{por ecuación (C.24)})$$

$$\frac{1}{X_n} = E \text{CET}_{.max.k} 1; nk// C encendido/ \quad (\text{por la ecuación (9.1))}.$$

Para aplicar la ecuación (C.24), nos basamos en  $X_k$  y  $T_{.max.k} 1; nk//$  siendo variables aleatorias independientes. El ejercicio 9.2-2 le pide que justifique esta afirmación.

Consideraremos la expresión  $\text{max.k} 1; nk//$ . Tenemos

$$\text{máx.k} 1; nk// D ( k 1 si k > dn=2e ; si.k dn=2e :$$

Si  $n$  es par, cada término desde  $T_{.dn=2e}$  hasta  $T_{.n}$  aparece exactamente dos veces en la suma, y si  $n$  es impar, todos estos términos aparecen dos veces y  $T_{.bn=2c}$  aparece una vez. Así, tenemos

$$E \text{CET}_{.n}/ = \frac{2}{n} \sum_{k=1}^{n/2} X_{D1} E \text{CET}_{.k} / C On/ : \quad kDbn=2c$$

Demostramos que  $E \text{CET}_{.n}/ D On/$  por sustitución. Suponga que  $E \text{CET}_{.n}/ cn$  para alguna constante  $c$  que satisface las condiciones iniciales de la recurrencia. Suponemos que  $T_{.n}/ DO.1/$  para  $n$  menor que alguna constante; seleccionaremos esta constante más adelante. También elegimos una constante  $a$  tal que la función descrita por el término  $On/$  anterior (que describe el componente no recursivo del tiempo de ejecución del algoritmo) esté limitada desde arriba por  $a$  para todo  $n>0$ . Usando esta hipótesis inductiva, tenemos

$$E \text{CET}_{.n}/ = \frac{2}{n} \sum_{k=1}^{n/2} X_{D1} ck \text{ puede } \quad kDbn=2c$$

$$= \frac{2c}{n} \sum_{k=1}^{n/2} X_{D1} k \quad \begin{matrix} bn=2c \\ X \\ kD1 \end{matrix} \quad k! Poder$$

$$\begin{array}{l}
 \text{D} \quad \frac{2c}{\text{norte}} \quad \frac{\cdot n 1/n}{2} \quad \frac{\cdot bn=2c 1/bn=2c}{2} \quad \text{Poder} \\
 \text{D} \quad \frac{2c}{\text{norte}} \quad \frac{\cdot n 1/n}{2} \quad \frac{\cdot n=2 2/n=2 1/}{2} \quad \text{Poder} \\
 \text{D} \quad \frac{2c}{\text{norte}} \quad \frac{n^2 \text{norte}}{2} \quad \frac{n^2=4 3n=2 C 2}{2} \quad \text{Poder} \\
 \text{D} \quad \frac{C}{\text{norte}} \quad \frac{3n^2}{4} \quad \frac{C}{2} \quad 2 \text{ latas} \\
 \text{re} \quad \frac{3n}{4} \quad \frac{C}{2} \quad \frac{1}{2} \quad \text{norte} \quad \text{Poder} \\
 \frac{3cn}{4} \quad C \quad \frac{C}{2} \quad \text{lata 2} \\
 \text{re cn} \quad \frac{cn}{4} \quad \frac{C}{2} \quad \text{un} \quad \dots
 \end{array}$$

Para completar la prueba, necesitamos mostrar que para  $n$  suficientemente grande, esta última expresión es como máximo  $cn$  o, de manera equivalente, que  $cn=4$   $c=2$  an 0. Si sumamos  $c=2$  a ambos lados y factorizamos  $n$ , obtenemos  $nc=4$   $a/c=2$ . Siempre que elijamos la constante  $c$  para que  $c=4 a>0$ , es decir,  $c > 4a$ , podemos dividir ambos lados por  $c=4 a$ , dando

$$\frac{c=2}{\text{norte}} \quad \frac{2c}{\text{c } 4a} \quad \dots$$

Así, si suponemos que  $T \cdot n / D O.1 /$  para  $n < 2c=c 4a/$ , entonces  $E \leq T \cdot n / D O_n/$ .

Concluimos que podemos encontrar cualquier estadístico de orden, y en particular la mediana, en el tiempo lineal esperado, asumiendo que los elementos son distintos.

### Ejercicios

#### 9.2-1

Demuestre que RANDOMIZED-SELECT nunca realiza una llamada recursiva a un arreglo de longitud 0.

#### 9.2-2

Argumente que el indicador variable aleatoria  $X_k$  y el valor  $T \cdot \max_k 1; nk//$  son independientes.

#### 9.2-3

Escriba una versión iterativa de SELECCIÓN ALEATORIA.

## 9.2-4

Supongamos que usamos RANDOMIZED-SELECT para seleccionar el elemento mínimo del arreglo AD h3; 2; 9; 0; 7; 5; 4; 8; 6; 1i. Describa una secuencia de particiones que resulte en el peor de los casos de RANDOMIZED-SELECT.

## 9.3 Selección en el peor de los casos de tiempo lineal

Ahora examinamos un algoritmo de selección cuyo tiempo de ejecución es  $O(n)$  en el peor de los casos. Al igual que RANDOMIZED-SELECT, el algoritmo SELECT encuentra el elemento deseado mediante la partición recursiva de la matriz de entrada. Aquí, sin embargo, garantizamos una buena división al particionar la matriz. SELECT usa el algoritmo de partición determinista PARTITION de quicksort (ver Sección 7.1), pero modificado para tomar el elemento para particionar como un parámetro de entrada.

El algoritmo SELECT determina el  $i$ -ésimo más pequeño de una matriz de entrada de  $n > 1$  elementos distintos mediante la ejecución de los siguientes pasos. (Si  $n = 1$ , entonces SELECT simplemente devuelve su único valor de entrada como el  $i$ -ésimo más pequeño).

1. Dividir los  $n$  elementos de la matriz de entrada en  $b_n = \lceil n/5 \rceil$  grupos de 5 elementos cada uno y como máximo un grupo formado por los  $n \bmod 5$  elementos restantes.
2. Encuentre la mediana de cada uno de los grupos  $d_n = \lceil 5/2 \rceil$  clasificando primero por inserción los elementos de cada grupo (de los cuales hay un máximo de 5) y luego eligiendo la mediana de la lista ordenada de elementos del grupo.
3. Usa SELECCIONAR recursivamente para encontrar la mediana  $x$  de las medianas  $d_n = \lceil 5/2 \rceil$  encontradas en el paso 2. (Si hay un número par de medianas, entonces, según nuestra convención,  $x$  es la mediana inferior).
4. Particionar el arreglo de entrada alrededor de la mediana de las medianas  $x$  usando la versión modificada de PARTICIÓN. Sea  $k$  uno más que el número de elementos en el lado inferior de la partición, de modo que  $x$  es el  $k$ -ésimo elemento más pequeño y hay  $nk$  elementos en el lado superior de la partición.
5. Si  $i \leq k$ , entonces devuelve  $x$ . De lo contrario, use SELECT recursivamente para encontrar el  $i$ -ésimo elemento más pequeño en el lado inferior si  $i < k$ , o el  $(i-k)/k$ -ésimo elemento más pequeño en el lado superior si  $i > k$ .

Para analizar el tiempo de ejecución de SELECT, primero determinamos un límite inferior en el número de elementos que son mayores que el elemento de partición  $x$ . La figura 9.1 nos ayuda a visualizar esta contabilidad. Al menos la mitad de las medianas encontradas en

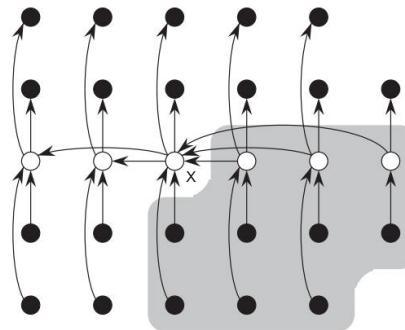


Figura 9.1 Análisis del algoritmo SELECT. Los  $n$  elementos están representados por pequeños círculos, y cada grupo de 5 elementos ocupa una columna. Las medianas de los grupos están blanqueadas y la mediana de las medianas  $x$  está etiquetada. (Al encontrar la mediana de un número par de elementos, usamos la mediana inferior). Las flechas van de los elementos más grandes a los más pequeños, de donde podemos ver que 3 de cada grupo completo de 5 elementos a la derecha de  $x$  son mayores que  $x$ , y 3 de cada grupo de 5 elementos a la izquierda de  $x$  son menores que  $x$ . Los elementos que se sabe que son mayores que  $x$  aparecen sobre un fondo sombreado.

paso 2 son mayores o iguales a la mediana de las medianas  $x$ .

<sup>1</sup> Así, al menos la mitad

de los grupos  $d_n=5e$  aportan al menos 3 elementos mayores que  $x$ , excepto el grupo que tiene menos de 5 elementos si 5 no divide  $n$  exactamente, y el grupo que contiene  $x$ . Descontando estos dos grupos, se sigue que el número de elementos mayor que  $x$  es al menos

$$3 \quad \frac{1}{2} \text{ en } 5m \quad 2 \quad \frac{3n}{10} \quad 6$$

De manera similar, al menos  $3n=10$  6 elementos son menores que  $x$ . Por lo tanto, en el peor de los casos, el paso 5 llama a SELECT recursivamente en un máximo de  $7n=10$  C 6 elementos.

Ahora podemos desarrollar una recurrencia para el tiempo de ejecución del peor de los casos  $T .n/$  del algoritmo SELECT. Los pasos 1, 2 y 4 se activan/tiempo. (El paso 2 consiste en llamadas On/ de orden de inserción en conjuntos de tamaño  $O.1/$ .) El paso 3 toma un tiempo  $T .dn=5e/$ , y el paso 5 toma un tiempo como máximo  $T .7n=10 C 6/$ , asumiendo que  $T$  es monótonamente creciente. Hacemos la suposición, que parece infundada al principio, que cualquier entrada de menos de 140 elementos requiere  $0.1/$  tiempo; el origen de la constante mágica 140 quedará claro en breve. Por lo tanto, podemos obtener la recurrencia

<sup>1</sup> Debido a nuestra suposición de que los números son distintos, todas las medianas excepto  $x$  son mayores o menor que  $x$ .

$$T_{.n/} \begin{cases} \dots & \text{si } n < 140 \\ (O_1 T_{.n/} dn=5e / CT .7n=10 C 6/ C On/ si n 140) & \end{cases}$$

Demostramos que el tiempo de ejecución es lineal por sustitución. Más específicamente, mostraremos que  $T_{.n/} \leq cn$  para alguna constante  $c$  suficientemente grande y todo  $n > 0$ .

Empezamos suponiendo que  $T_{.n/} \leq cn$  para alguna constante  $c$  suficientemente grande y todo  $n < 140$ ; esta suposición se cumple si  $c$  es lo suficientemente grande. También elegimos una constante  $a$  tal que la función descrita por el término  $On/$  anterior (que describe el componente no recursivo del tiempo de ejecución del algoritmo) esté acotada arriba por  $an$  para todo  $n > 0$ . Sustituyendo esta hipótesis inductiva en el lado derecho de la recurrencia se obtiene

$$\begin{aligned} T_{.n/} &\leq c dn=5e C c.7n=10 C 6/ C an cn=5 \\ &\leq C c C 7cn=10 C 6c C an D 9cn=10 \\ &\leq C 7c C an D cn C .cn=10 \\ &\leq C 7c C an/ ; \end{aligned}$$

que es como mucho  $cn$  si

$$cn=10 C 7c C an 0 \quad (9.2)$$

La desigualdad (9.2) es equivalente a la desigualdad  $c 10a.n=.n 70//$  cuando  $n > 70$ .

Dado que suponemos que  $n \geq 140$ , tenemos  $n = .n 70/ 2$ , por lo que elegir  $c 20a$  satisfará la desigualdad (9.2). (Tenga en cuenta que la constante 140 no tiene nada de especial; podríamos reemplazarla por cualquier número entero estrictamente mayor que 70 y luego elegir  $c$  en consecuencia). El tiempo de ejecución del peor caso de SELECT es, por lo tanto, lineal.

Como en una ordenación por comparación (ver Sección 8.1), SELECT y RANDOMIZED-SELECT determinan información sobre el orden relativo de los elementos solo comparando elementos. Recuerde del capítulo 8 que la clasificación requiere  $.n \lg n/$  de tiempo en el modelo de comparación, incluso en promedio (vea el problema 8-1). Los algoritmos de clasificación de tiempo lineal del Capítulo 8 hacen suposiciones sobre la entrada. Por el contrario, los algoritmos de selección de tiempo lineal de este capítulo no requieren ninguna suposición sobre la entrada. No están sujetos al límite inferior  $.n \lg n/$  porque logran resolver el problema de selección sin ordenar. Por lo tanto, resolver el problema de selección ordenando e indexando, como se presentó en la introducción de este capítulo, es asintóticamente ineficiente.

## Ejercicios

### 9.3-1

En el algoritmo SELECT, los elementos de entrada se dividen en grupos de 5. ¿Funcionará el algoritmo en tiempo lineal si se dividen en grupos de 7? Argumente que SELECT no se ejecuta en tiempo lineal si se usan grupos de 3.

### 9.3-2

Analice SELECT para mostrar que si  $n \geq 140$ , entonces al menos  $\lceil n/4 \rceil$  elementos son mayores que la mediana de las medianas  $x$  y al menos  $\lceil n/4 \rceil$  elementos son menores que  $x$ .

### 9.3-3

Muestre cómo se puede hacer que quicksort se ejecute en  $O(n \lg n)$  time en el peor de los casos, suponiendo que todos los elementos son distintos.

### 9.3-4 ?

Suponga que un algoritmo usa solo comparaciones para encontrar el  $i$ -ésimo elemento más pequeño en un conjunto de  $n$  elementos. Muestre que también puede encontrar los  $i+1$  elementos más pequeños y los  $n-i$  elementos más grandes sin realizar comparaciones adicionales.

### 9.3-5

Suponga que tiene una subrutina de mediana de tiempo lineal de "caja negra" en el peor de los casos. Proporcione un algoritmo simple de tiempo lineal que resuelva el problema de selección para una estadística de orden arbitrario.

### 9.3-6

Los  $k$ -ésimos cuantiles de un conjunto de  $n$  elementos son las estadísticas de  $k$  orden que dividen el conjunto ordenado en  $k$  conjuntos de igual tamaño (con una precisión de 1). Proporcione un algoritmo  $O(n \lg k)$ -time para enumerar los  $k$ -ésimos cuantiles de un conjunto.

### 9.3-7

Describa un algoritmo  $O(n)$ -time que, dado un conjunto  $S$  de  $n$  números distintos y un entero positivo  $kn$ , determine los  $k$  números en  $S$  que están más cerca de la mediana de  $S$ .

### 9.3-8

Sean  $X[1:n] \times Y[1:n]$  dos arreglos, cada uno de los cuales contiene  $n$  números ya ordenados. Proporcione un algoritmo  $O(n \lg n)$ -time para encontrar la mediana de todos los  $2n$  elementos en las matrices  $X$  e  $Y$ .

### 9.3-9

El profesor Olay es consultor de una compañía petrolera, que está planeando un gran oleoducto que corre de este a oeste a través de un campo petrolero de  $n$  pozos. La empresa quiere conectarse

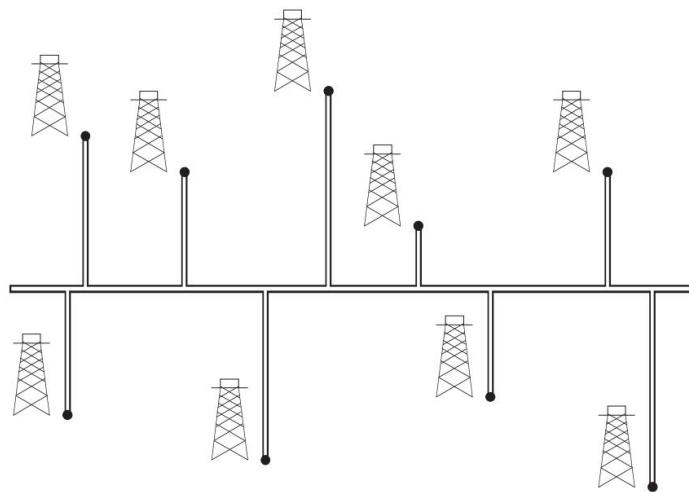


Figura 9.2 El profesor Olay necesita determinar la posición del oleoducto este-oeste que minimiza la longitud total de las estribaciones norte-sur.

una tubería secundaria desde cada pozo directamente a la tubería principal a lo largo de la ruta más corta (ya sea al norte o al sur), como se muestra en la Figura 9.2. Dadas las coordenadas  $x$  e  $y$  de los pozos, ¿cómo debería elegir el profesor la ubicación óptima de la tubería principal, que sería la que minimiza la longitud total de las estribaciones? Muestre cómo determinar la ubicación óptima en tiempo lineal.

## Problemas

**9-1 Números  $i$  más grandes en orden dado un conjunto de  $n$  números,** deseamos encontrar el  $i$  más grande en orden ordenado usando un algoritmo basado en comparación. Encuentre el algoritmo que implementa cada uno de los siguientes métodos con el mejor tiempo de ejecución asintótico en el peor de los casos y analice los tiempos de ejecución de los algoritmos en términos de  $n$  e  $i$ .

- Ordena los números y enumera la  $i$  más grande.
- Cree una cola de máxima prioridad a partir de los números y llame a EXTRACT-MAX  $i$  veces.
- Use un algoritmo estadístico de orden para encontrar el  $i$ -ésimo número más grande, divida alrededor ese número, y ordenar los  $i$  números más grandes.

### 9-2 mediana ponderada

Para  $n$  elementos distintos  $x_1; x_2; \dots; x_n$  con pesos positivos  $w_1; w_2; \dots; w_n$  tal que  $\sum i D 1 w_i D 1$ , la mediana ponderada (inferior) es el elemento  $x_k$  que satisface

$$\frac{\sum_{i=1}^k w_i}{2} < \frac{1}{2}$$

$x_i < x_k$

$$\frac{\sum_{i=1}^{k-1} w_i}{2} < \frac{1}{2} \leq \frac{\sum_{i=1}^k w_i}{2} < \frac{1}{2} + \frac{\sum_{i=k+1}^n w_i}{2}$$

$x_i > x_k$

Por ejemplo, si los elementos son  $0: 1; 0: 35; 0: 05; 0: 1; 0: 15; 0: 05; 0: 2$  y cada elemento es igual a su peso (es decir,  $w_i = x_i$  para  $i \in \{1, 2, \dots, 7\}$ ), entonces la mediana es  $0: 1$ , pero la mediana ponderada es  $0: 2$ .

- a. Argumente que la mediana de  $x_1; x_2; \dots; x_n$  es la mediana ponderada de las  $x_i$  con pesos  $w_i = 1/n$  para  $i \in \{1, 2, \dots, n\}$ .
- b. Muestre cómo calcular la mediana ponderada de  $n$  elementos en  $O(n \lg n)$  tiempo en el peor caso de tiempo utilizando la clasificación.
- c. Muestre cómo calcular la mediana ponderada en  $O(n)$  tiempo en el peor de los casos usando un algoritmo de mediana de tiempo lineal como SELECT de la Sección 9.3.

El problema de ubicación de la oficina de correos se define de la siguiente manera. Nos dan  $n$  puntos  $p_1; p_2; \dots; p_n$  con pesos asociados  $w_1; w_2; \dots; w_n$ . Deseamos encontrar un punto  $p$  (no necesariamente uno de los puntos de entrada) que minimice la suma  $\sum_{i=1}^n w_i d(p, p_i)$ , donde  $d(a, b)$  es la distancia entre los puntos  $a$  y  $b$ .

- d. Argumente que la mediana ponderada es la mejor solución para el problema de ubicación de la oficina postal unidimensional, en el que los puntos son simplemente números reales y la distancia entre los puntos  $a$  y  $b$  es  $|a - b|$ .
- e. Encuentre la mejor solución para el problema de ubicación de la oficina de correos en 2 dimensiones, en el que los puntos son  $(x_i, y_i)$  pares de coordenadas y la distancia entre puntos  $a$  y  $b$  es la distancia Manhattan dada por  $d(a, b) = |x_a - x_b| + |y_a - y_b|$ .

### 9-3 Estadísticas de orden

pequeño Mostramos que el número  $T(n)$  del peor caso de las comparaciones utilizadas por SELECT para seleccionar la estadística de orden  $i$  entre  $n$  números satisface  $T(n) = O(n)$ , pero la constante oculta por  $C$  es bastante grande. Cuando  $i$  es pequeño en relación con  $n$ , podemos implementar un procedimiento diferente que use SELECT como una subrutina pero haga menos comparaciones en el peor de los casos.

- a. Describa un algoritmo que use comparaciones  $U_i/n$  para encontrar el  $i$ -ésimo más pequeño de  $n$  elementos, donde

si en=2;

$U_i.n / D (T.bn=2c \wedge U_i.bn=2e) / CT .2i$  / de lo contrario:

(Sugerencia: Comience con  $b_n=2c$  comparaciones separadas por pares y recurra al conjunto que contiene el elemento más pequeño de cada par).

- b. Demuestre que, si  $i < n=2$ , entonces  $\cup_{j=n}^i D_j \subseteq OT_{\leq i}$ .

C. Muestre que si  $i$  es una constante menor que  $n=2$ , entonces  $U_i.n / D \in C.O.lg n$ .

d. Muestre que si  $i \in D$  para  $n=k$  para  $k \geq 2$ , entonces  $U_i \in D$  para  $n \in C \setminus OT$ .  $2n=k/lq$   $k/l$ .

9-4 Análisis alternativo de selección aleatoria En este problema, usamos variables aleatorias indicadoras para analizar el procedimiento SELECCIÓN ALEATORIA de manera similar a nuestro análisis de RANDOMIZED-QUICKSORT en la Sección 7.4.2.

Al igual que en el análisis de clasificación rápida, asumimos que todos los elementos son distintos y renombramos los elementos de la matriz de entrada A como '1'; '2';...; 'n', donde 'i' es el i-ésimo elemento más pequeño. Así, la llamada RANDOMIZED-SELECT.A; 1; norte; k devuelve 'k'.

Para 1 i<jn, sea

Xijk DI f'i se compara con 'j en algún momento durante la ejecución del algoritmo para encontrar 'kg ;

- a. Dé una expresión exacta para  $E \propto X_{ijk}$ . (Sugerencia: su expresión puede diferir dependiendo de los valores de  $i$ ,  $j$  y  $k$ .)
  - b. Sea  $X_k$  el número total de comparaciones entre elementos del arreglo A al encontrar ' $k$ '. Muestra esa

$$E \in \mathbb{X}^k \times \frac{1}{\sum_{i=1}^k C_i D_i} \mathbb{C}^{X_n} \quad \frac{\sum_{i=1}^k \sum_{j=1}^{C_i} C_j}{\sum_{i=1}^k D_i} \mathbb{C}^X \quad \frac{k^2}{\sum_{i=1}^k C_i} \mathbb{C}^{X_n}$$

C. Demuestre que  $E \subset X_k$  4n.

- d. Concluya que, asumiendo que todos los elementos de la matriz A son distintos, RANDOMIZED SELECT se ejecuta en el tiempo esperado en:

---

## Notas del capítulo

El algoritmo de búsqueda de la mediana de tiempo lineal en el peor de los casos fue ideado por Blum, Floyd, Pratt, Rivest y Tarjan [50]. La versión aleatoria rápida se debe a Hoare [169]. Floyd y Rivest [108] han desarrollado una versión aleatoria mejorada que divide alrededor de un elemento seleccionado recursivamente de una pequeña muestra de los elementos.

Todavía se desconoce exactamente cuántas comparaciones se necesitan para determinar la mediana. Bent y John [41] dieron un límite inferior de  $2n$  comparaciones para encontrar la mediana, y Schönhage, Paterson y Pippenger [302] dieron un límite superior de  $3n$ . Dor y Zwick han mejorado en ambos límites. Su límite superior [93] es ligeramente inferior a  $2.95n$ , y su límite inferior [94] es  $0.2 C/n$ , para una pequeña constante positiva, mejorando así ligeramente el trabajo relacionado de Dor et al. [92].

Paterson [272] describe algunos de estos resultados junto con otros trabajos relacionados.

---

### III Estructuras de datos

---

## Introducción

Los conjuntos son tan fundamentales para la informática como lo son para las matemáticas. Mientras que los conjuntos matemáticos no cambian, los conjuntos manipulados por algoritmos pueden crecer, reducirse o cambiar de otro modo con el tiempo. A tales conjuntos los llamamos dinámicos. Los siguientes cinco capítulos presentan algunas técnicas básicas para representar conjuntos dinámicos finitos y manipularlos en una computadora.

Los algoritmos pueden requerir que se realicen varios tipos diferentes de operaciones en conjuntos. Por ejemplo, muchos algoritmos solo necesitan la capacidad de insertar elementos, eliminar elementos y probar la pertenencia a un conjunto. A un conjunto dinámico que soporta estas operaciones lo llamamos diccionario. Otros algoritmos requieren operaciones más complicadas. Por ejemplo, las colas de prioridad mínima, que el Capítulo 6 presentó en el contexto de la estructura de datos del montón, admiten las operaciones de insertar un elemento y extraer el elemento más pequeño de un conjunto. La mejor manera de implementar un conjunto dinámico depende de las operaciones que deben admitirse.

### Elementos de un conjunto dinámico

En una implementación típica de un conjunto dinámico, cada elemento está representado por un objeto cuyos atributos pueden examinarse y manipularse si tenemos un puntero al objeto. (La sección 10.3 analiza la implementación de objetos y punteros en entornos de programación que no los contienen como tipos de datos básicos). Algunos tipos de conjuntos dinámicos asumen que uno de los atributos del objeto es una clave de identificación . Si las claves son todas diferentes, podemos pensar en el conjunto dinámico como un conjunto de valores clave. El objeto puede contener datos de satélite, que se transportan en otros atributos del objeto pero que, por lo demás, no son utilizados por la implementación del conjunto. Puede

también tienen atributos que son manipulados por las operaciones de conjunto; estos atributos pueden contener datos o punteros a otros objetos del conjunto.

Algunos conjuntos dinámicos presuponen que las claves se extraen de un conjunto totalmente ordenado, como los números reales, o el conjunto de todas las palabras bajo el orden alfabético habitual. Una ordenación total nos permite definir el elemento mínimo del conjunto, por ejemplo, o hablar del siguiente elemento más grande que un elemento dado en un conjunto.

#### Operaciones sobre conjuntos dinámicos

Las operaciones en un conjunto dinámico se pueden agrupar en dos categorías: consultas, que simplemente devuelven información sobre el conjunto, y operaciones de modificación, que cambian el conjunto. Aquí hay una lista de operaciones típicas. Cualquier aplicación específica generalmente requerirá solo algunos de estos para implementarse.

##### BUSCAR.S; k/ Una

consulta que, dado un conjunto S y un valor clave k, devuelve un puntero x a un elemento en S tal que x:clave D k, o NIL si ningún elemento pertenece a S.

##### INSERTAR.S; x/

Una operación de modificación que aumenta el conjunto S con el elemento señalado por x. Por lo general, asumimos que cualquier atributo en el elemento x que necesite la implementación del conjunto ya se ha inicializado.

##### ELIMINAR.S; x/

Una operación de modificación que, dado un puntero x a un elemento en el conjunto S, quita x de S. (Tenga en cuenta que esta operación lleva un puntero a un elemento x, no un valor clave).

##### MINIMUM.S / Una

consulta sobre un conjunto S totalmente ordenado que devuelve un puntero al elemento de S con la clave más pequeña.

##### MAXIMUM.S / Una

consulta sobre un conjunto S totalmente ordenado que devuelve un puntero al elemento de S con la clave más grande.

##### SUCESOR.S; x/ Una

consulta que, dado un elemento x cuya clave es de un conjunto S totalmente ordenado, devuelve un puntero al siguiente elemento más grande en S, o NIL si x es el elemento máximo.

##### ANTECESORES; x/ Una

consulta que, dado un elemento x cuya clave es de un conjunto S totalmente ordenado, devuelve un puntero al siguiente elemento más pequeño en S, o NIL si x es el elemento mínimo.

En algunas situaciones, podemos extender las consultas SUCCESSOR y PREDECESSOR para que se apliquen a conjuntos con claves no distintas. Para un conjunto de  $n$  claves, la presunción normal es que una llamada a MINIMUM seguida de  $n - 1$  llamadas a SUCCESSOR enumera los elementos del conjunto en orden ordenado.

Por lo general, medimos el tiempo necesario para ejecutar una operación de conjunto en términos del tamaño del conjunto. Por ejemplo, el Capítulo 13 describe una estructura de datos que puede admitir cualquiera de las operaciones enumeradas anteriormente en un conjunto de tamaño  $n$  en el tiempo  $O.\lg n$ .

### Resumen de la Parte III

Los capítulos 10 a 14 describen varias estructuras de datos que podemos usar para implementar conjuntos dinámicos; usaremos muchos de estos más adelante para construir algoritmos eficientes para una variedad de problemas. Ya vimos otra estructura de datos importante, el montón, en el Capítulo 6.

El Capítulo 10 presenta los elementos esenciales para trabajar con estructuras de datos simples, como pilas, colas, listas enlazadas y árboles enraizados. También muestra cómo implementar objetos y punteros en entornos de programación que no los admiten como primitivos. Si ha tomado un curso de introducción a la programación, gran parte de este material le resultará familiar.

El Capítulo 11 presenta las tablas hash, que admiten las operaciones de diccionario IN SERT, DELETE y SEARCH. En el peor de los casos, el hash requiere  $\sqrt{n}$  tiempo para realizar una operación de BÚSQUEDA, pero el tiempo esperado para las operaciones de tabla hash es  $O.1$ .

El análisis de hash se basa en la probabilidad, pero la mayor parte del capítulo no requiere conocimientos previos sobre el tema.

Los árboles de búsqueda binarios, que se tratan en el Capítulo 12, admiten todas las operaciones de conjuntos dinámicos enumeradas anteriormente. En el peor de los casos, cada operación toma  $\sqrt{n}$  tiempo en un árbol con  $n$  elementos, pero en un árbol de búsqueda binaria construido al azar, el tiempo esperado para cada operación es  $O.\lg n$ . Los árboles de búsqueda binarios sirven como base para muchas otras estructuras de datos.

El Capítulo 13 presenta los árboles rojo-negro, que son una variante de los árboles de búsqueda binarios. A diferencia de los árboles de búsqueda binarios ordinarios, se garantiza que los árboles rojo-negro funcionan bien: las operaciones tardan  $O.\lg n$  en el peor de los casos. Un árbol rojo-negro es un árbol de búsqueda equilibrado; El Capítulo 18 de la Parte V presenta otro tipo de árbol de búsqueda equilibrado, llamado árbol B. Aunque la mecánica de los árboles rojo-negros es un tanto complicada, puedes deducir la mayoría de sus propiedades del capítulo sin estudiar la mecánica en detalle. Sin embargo, probablemente encontrará bastante instructivo recorrer el código.

En el Capítulo 14, mostramos cómo aumentar los árboles rojo-negro para admitir operaciones distintas de las básicas enumeradas anteriormente. Primero, los aumentamos para que podamos mantener dinámicamente estadísticas de pedidos para un conjunto de claves. Luego, los aumentamos de una manera diferente para mantener los intervalos de los números reales.

---

## 10

## Estructuras de datos elementales

En este capítulo, examinamos la representación de conjuntos dinámicos mediante estructuras de datos simples que utilizan punteros. Aunque podemos construir muchas estructuras de datos complejas utilizando punteros, presentamos solo las rudimentarias: pilas, colas, listas enlazadas y árboles enraizados. También mostramos formas de sintetizar objetos y punteros de ar rayos

---

### 10.1 Pilas y colas

Las pilas y las colas son conjuntos dinámicos en los que se especifica previamente el elemento eliminado del conjunto por la operación `DELETE`. En una pila, el elemento eliminado del conjunto es el insertado más recientemente: la pila implementa una política de último en entrar, primero en salir o `LIFO`. De manera similar, en una cola, el elemento eliminado es siempre el que ha estado en el conjunto durante más tiempo: la cola implementa una política de primero en entrar, primero en salir o `FIFO`. Hay varias formas eficientes de implementar pilas y colas en una computadora. En esta sección mostramos cómo usar una matriz simple para implementar cada uno.

#### pilas

La operación `INSERT` en una pila a menudo se llama `PUSH`, y la operación `DELETE`, que no toma un argumento de elemento, a menudo se llama `POP`. Estos nombres son alusiones a pilas físicas, como las pilas de platos con resorte que se usan en las cafeterías. El orden en que se sacan las placas de la pila es el orden inverso al que se colocaron en la pila, ya que solo se puede acceder a la placa superior.

Como muestra la figura 10.1, podemos implementar una pila de  $n$  elementos como máximo con un arreglo `S[0 : n]`. La matriz tiene un atributo `S.top` que indexa el más reciente

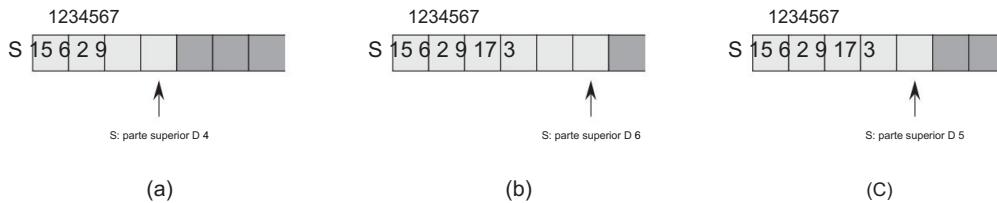


Figura 10.1 Una implementación de matriz de una pila S. Los elementos de la pila aparecen solo en las posiciones ligeramente sombreadas. (a) La pila S tiene 4 elementos. El elemento superior es 9. (b) Pila S después de las llamadas PUSH.S; 17 y PUSH.S; 3. (c) La pila S después de que la llamada POP.S / haya devuelto el elemento 3, que es el que se envió más recientemente. Aunque el elemento 3 todavía aparece en la matriz, ya no está en la pila; la parte superior es el elemento 17.

elemento insertado. La pila consta de elementos  $SCE1 : S:\text{top}$ , donde  $SCE1$  es el elemento en la parte inferior de la pila y  $S:\text{top}$  es el elemento en la parte superior.

Cuando  $S:\text{top } D \ 0$ , la pila no contiene elementos y está vacía. Podemos probar para ver si la pila está vacía mediante la operación de consulta STACK-EMPTY. Si intentamos abrir una pila vacía, decimos que la pila se desborda, lo que normalmente es un error.

Si  $S:\text{top}$  excede  $n$ , la pila se desborda. (En nuestra implementación de pseudocódigo, no nos preocupamos por el desbordamiento de pila).

Podemos implementar cada una de las operaciones de la pila con solo unas pocas líneas de código:

STACK-EMPTY.S / 1 si  
S:superior == 0 2  
devuelve VERDADERO 3 de  
lo contrario devuelve FALSO

```
PUSH.S; x/ 1  
  
S:superior D S:superior C 1 2  
SŒS:superior D x  
  
POP.S / 1  
  
si STACK-EMPTY.S / 2 error  
"underflow" 3 else S:top D S:top 1 4  
  
return SŒS:top C 1
```

La figura 10.1 muestra los efectos de las operaciones de modificación PUSH y POP. Cada una de las tres operaciones de pila toma  $O(1)$  tiempo.

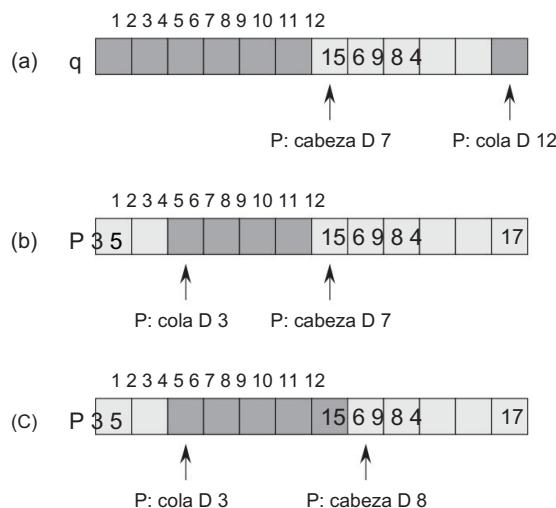


Figura 10.2 Una cola implementada usando una matriz QŒ1 :: 12. Los elementos de la cola aparecen solo en las posiciones ligeramente sombreadas. (a) La cola tiene 5 elementos, en las ubicaciones QŒ7 :: 11. (b) La configuración de la cola después de las llamadas ENQUEUE.Q; 17/, ENQUEUE.Q; 3/, y ENQUEUE.Q; 5/. (c) La configuración de la cola después de la llamada DEQUEUE.Q/ devuelve el valor de clave 15 anteriormente en la cabecera de la cola. La nueva cabeza tiene clave 6.

## Colas

Llamamos a la operación INSERT en una cola ENQUEUE, y llamamos a la operación DELETE DEQUEUE; al igual que la operación de pila POP, DEQUEUE no toma ningún argumento de elemento. La propiedad FIFO de una cola hace que opere como una fila de clientes esperando para pagar en un cajero. La cola tiene cara y cola. Cuando un elemento se pone en cola, ocupa su lugar al final de la cola, al igual que un cliente recién llegado ocupa un lugar al final de la fila. El elemento eliminado es siempre el que está al principio de la cola, como el cliente al principio de la fila que ha esperado más tiempo.

La figura 10.2 muestra una forma de implementar una cola de como máximo n 1 elementos utilizando un arreglo QŒ1 :: n. La cola tiene un atributo Q:cabeza que indexa o apunta a su cabeza. El atributo Q:tail indexa la siguiente ubicación en la que se insertará en la cola un elemento recién llegado. Los elementos de la cola residen en las ubicaciones Q:head; P: cabeza C 1; :: ; P: cola 1, donde "envolvemos" en el sentido de que la ubicación 1 sigue inmediatamente a la ubicación n en un orden circular. Cuando Q:head D Q:tail, la cola está vacía. Inicialmente, tenemos Q:cabeza D Q:cola D 1. Si intentamos sacar de la cola un elemento de una cola vacía, la cola se desborda.

Cuando Q:head D Q:tail C 1, la cola está llena, y si intentamos poner en cola un elemento, entonces la cola se desborda.

En nuestros procedimientos ENQUEUE y DEQUEUE, hemos omitido la comprobación de errores por subdesbordamiento y desbordamiento. (El ejercicio 10.1-4 le pide que proporcione un código que verifique estas dos condiciones de error). El pseudocódigo asume que n D Q:longitud.

```

ENQUEUE.Q; x/ 1
Q<EQ:cola D x 2 if
Q:cola == Q:longitud 3 Q:cola
D 1 4 else Q:cola D
Q:cola C 1

DEQUEUE.Q/ 1
x D Q<EQ:head 2 if
Q:head == Q:longitud 3 Q:head
D 1 4 else Q:head D
Q:head C 1 5 return x

```

La figura 10.2 muestra los efectos de las operaciones ENQUEUE y DEQUEUE . Cada operación toma O.1/ tiempo.

### Ejercicios

#### 10.1-1

Usando la figura 10.1 como modelo, ilustre el resultado de cada operación en la secuencia PUSH.S; 4/, PULSADORES; 1/, PULS.S; 3/, POP.S /, PUSH.S; 8/, y POP.S / en una pila S inicialmente vacía almacenada en el arreglo SŒ1 :: 6.

#### 10.1-2

Explique cómo implementar dos pilas en una matriz AŒ1 :: n de tal manera que ninguna pila se desborde a menos que el número total de elementos en ambas pilas juntas sea n. Las operaciones PUSH y POP deben ejecutarse en tiempo O.1/.

#### 10.1-3

Usando la Figura 10.2 como modelo, ilustre el resultado de cada operación en la secuencia ENQUEUE.Q; 4/, ENQUEUE.Q; 1/, ENQUEUE.Q; 3/, DESCARGAR.Q/, ENCARGAR.Q; 8/ y DEQUEUE.Q/ en una cola inicialmente vacía Q almacenada en el arreglo QŒ1 :: 6.

#### 10.1-4

Vuelva a escribir ENQUEUE y DEQUEUE para detectar el desbordamiento y el desbordamiento de una cola.

## 10.1-5

Mientras que una pila permite la inserción y eliminación de elementos en un solo extremo, y una cola permite la inserción en un extremo y la eliminación en el otro extremo, una deque (cola de dos extremos) permite la inserción y eliminación en ambos extremos. Escriba cuatro procedimientos O(1)-time para insertar elementos y eliminar elementos de ambos extremos de una deque implementada por una matriz.

## 10.1-6

Muestre cómo implementar una cola usando dos pilas. Analizar el tiempo de ejecución de las operaciones en cola.

## 10.1-7

Muestre cómo implementar una pila utilizando dos colas. Analizar el tiempo de ejecución de las operaciones de la pila.

---

## 10.2 Listas enlazadas

Una lista enlazada es una estructura de datos en la que los objetos se organizan en un orden lineal. Sin embargo, a diferencia de una matriz, en la que el orden lineal está determinado por los índices de la matriz, el orden en una lista enlazada está determinado por un puntero en cada objeto. Las listas enlazadas brindan una representación simple y flexible para conjuntos dinámicos, que admiten (aunque no necesariamente de manera eficiente) todas las operaciones enumeradas en la página 230.

Como se muestra en la figura 10.3, cada elemento de una lista L doblemente enlazada es un objeto con una clave de atributo y otros dos atributos de puntero: siguiente y pre. El objeto también puede contener otros datos satelitales. Dado un elemento x en la lista, x:next apunta a su sucesor en la lista enlazada y x:pre apunta a su predecesor. Si x:pre D NIL, el elemento x no tiene predecesor y por lo tanto es el primer elemento, o cabeza, de la lista. Si x:siguiente D NIL, el elemento x no tiene sucesor y por lo tanto es el último elemento, o cola, de la lista. Un atributo L:head apunta al primer elemento de la lista. Si L:head D NIL, la lista está vacía.

Una lista puede tener una de varias formas. Puede ser un enlace simple o doble, puede estar ordenado o no, y puede ser circular o no. Si una lista está enlazada individualmente, omitimos el puntero pre en cada elemento. Si se ordena una lista, el orden lineal de la lista corresponde al orden lineal de las claves almacenadas en los elementos de la lista; el elemento mínimo es entonces la cabeza de la lista y el elemento máximo es la cola. Si la lista no está ordenada, los elementos pueden aparecer en cualquier orden. En una lista circular, el prepuntero de la cabeza de la lista apunta a la cola, y el siguiente puntero de la cola de la lista apunta a la cabeza. Podemos pensar en una lista circular como un anillo de

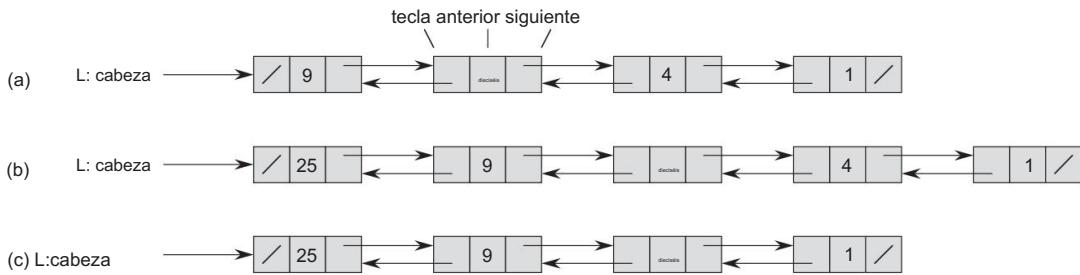


Figura 10.3 (a) Una lista doblemente enlazada L que representa el conjunto dinámico f1; 4; 9; 16 g. Cada elemento de la lista es un objeto con atributos para la clave y punteros (mostrados por flechas) al objeto siguiente y anterior. El siguiente atributo de la cola y el preatributo de la cabeza son NIL, indicados por una barra diagonal. El atributo L:head apunta a la cabeza. (b) Despu  s de la ejecuci  n de LIST-INSERT.L; x/, donde x:clave D 25, la lista enlazada tiene un nuevo objeto con la clave 25 como nuevo encabezado. Este nuevo objeto apunta al antiguo encabezado con la clave 9. (c) El resultado de la siguiente llamada LIST-DELETE.L; x/, donde x apunta al objeto con la tecla 4.

elementos. En el resto de esta secci  n, suponemos que las listas con las que estamos trabajando no est  n ordenadas y est  n doblemente vinculadas.

#### B  squeda de una lista enlazada

El procedimiento LIST-SEARCH.L; k/ encuentra el primer elemento con clave k en la lista L mediante una b  squeda lineal simple, devolviendo un puntero a este elemento. Si no aparece ning  n objeto con la clave k en la lista, el procedimiento devuelve NIL. Para la lista enlazada de la figura 10.3(a), la llamada LIST-SEARCH.L; 4/ devuelve un puntero al tercer elemento, y la llamada LIST-SEARCH.L; 7/ devuelve NIL.

```

LISTA-BUSQUEDA.L; k/
1 x D L:head 2
mientras que x ≠ NIL y x:key ≠ k 3
    x D x:siguiente
4 volver x

```

Para buscar una lista de n objetos, el procedimiento B  SQUEDA EN LISTA toma ..n/ tiempo en el peor de los casos, ya que puede tener que buscar en toda la lista.

#### Insertar en una lista enlazada

Dado un elemento x cuyo atributo clave ya se ha establecido, el procedimiento LIST-INSERT "empalma" x en el frente de la lista enlazada, como se muestra en la Figura 10.3(b).

```

LISTA-INSERTAR.L; x/
1 x:siguiente D L:cabeza
2 si L:cabeza ≠ NIL 3
L:cabeza:pre D x 4 L:cabeza D
x 5 x:pre D NIL

```

(Recuerde que nuestra notación de atributos puede funcionar en cascada, de modo que L:head:pre denota el atributo pre del objeto al que apunta L:head ). El tiempo de ejecución de LIST INSERT en una lista de n elementos es O.1/.

#### Eliminar de una lista enlazada

El procedimiento LIST-DELETE elimina un elemento x de una lista enlazada L. Se le debe dar un puntero a x, y luego "empalma" x fuera de la lista actualizando los punteros.

Si deseamos eliminar un elemento con una clave dada, primero debemos llamar LIST-SEARCH para recuperar un puntero al elemento.

```

LISTA-BORRAR.L; x/ 1
si x:pre ≠ NIL 2
x:pre:siguiente D x:siguiente 3 else
L:cabeza D x:siguiente
4 si x:siguiente ≠ NIL
5      x:siguiente:pre D x:pre

```

La figura 10.3(c) muestra cómo se elimina un elemento de una lista enlazada. LIST-DELETE se ejecuta en tiempo O.1/, pero si deseamos eliminar un elemento con una clave dada, se requiere tiempo ,n/ en el peor de los casos porque primero debemos llamar a LIST-SEARCH para encontrar el elemento.

#### centinelas

El código para LIST-DELETE sería más simple si pudieramos ignorar las condiciones de contorno al principio y al final de la lista:

```

LISTA-BORRAR0 .L; x/
1 x:pre:siguiente D x:siguiente
2 x:siguiente:pre D x:pre

```

Un centinela es un objeto ficticio que nos permite simplificar las condiciones de contorno. Por ejemplo, supongamos que proporcionamos con la lista L un objeto L:nil que representa NIL

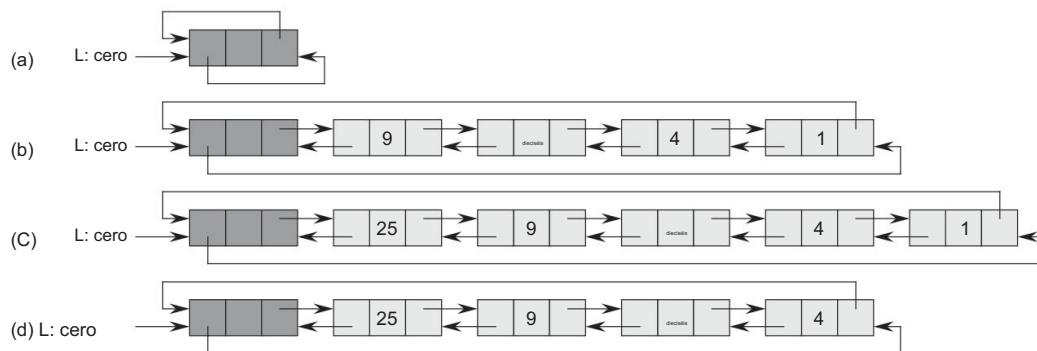


Figura 10.4 Una lista circular doblemente enlazada con un centinela. El centinela L:nil aparece entre la cabeza y la cola. El atributo L:head ya no es necesario, ya que podemos acceder al encabezado de la lista mediante L:nil:next. (a) Una lista vacía. (b) La lista enlazada de la figura 10.3(a), con la clave 9 al principio y la clave 1 al final. (c) La lista después de ejecutar LIST-INSERTO .L; x/, donde x:clave D 25. El nuevo objeto se convierte en el encabezado de la lista. (d) La lista después de eliminar el objeto con la clave 1. La nueva cola es el objeto con la clave 4.

pero tiene todos los atributos de los otros objetos en la lista. Siempre que tengamos una referencia a NIL en el código de la lista, la reemplazamos por una referencia al centinela L:nil. Como se muestra en la figura 10.4, este cambio convierte una lista regular doblemente enlazada en una lista circular doblemente enlazada con un centinela, en el que el centinela L:nil se encuentra entre la cabeza y la cola. El atributo L:nil:next apunta a la cabeza de la lista y L:nil:pre apunta a la cola. De manera similar, tanto el siguiente atributo de la cola como el preatributo de la cabeza apuntan a L:nil. Dado que L:nil:next apunta a la cabeza, podemos eliminar el atributo L:head por completo, reemplazando las referencias a él por referencias a L:nil:next. La figura 10.4(a) muestra que una lista vacía consiste solo en el centinela, y tanto L:nil:next como L:nil:pre apuntan a L:nil.

El código para LIST-SEARCH sigue siendo el mismo que antes, pero con las referencias a NIL y L:head cambiadas como se especifica arriba:

```

LISTA-BUSQUEDA0 .L; k/
1 x D L:nil:siguiente 2
mientras que x ≠ L:nil y x:tecla ≠ k 3
    x D x:siguiente
4 volver x

```

Usamos el procedimiento de dos líneas LIST-DELETE0 anterior para eliminar un elemento de la lista. El siguiente procedimiento inserta un elemento en la lista:

```

LISTA-INSERTAR0 .L; x/
1 x:siguiente D L:nil:siguiente
2 L:nil:siguiente:pre D x 3
L:nil:siguiente D x 4
x:pre D L:nil

```

La figura 10.4 muestra los efectos de LIST-INSERT0 y LIST-DELETE0 en una lista de muestra.

Los centinelas rara vez reducen los límites de tiempo asintóticos de las operaciones de estructura de datos, pero pueden reducir los factores constantes. La ventaja de usar centinelas dentro de los bucles suele ser una cuestión de claridad del código más que de velocidad; el código de la lista enlazada, por ejemplo, se vuelve más simple cuando usamos centinelas, pero ahorraremos solo  $O(1)$  tiempo en los procedimientos LIST-INSERT0 y LIST-DELETE0. En otras situaciones, sin embargo, el uso de centinelas ayuda a ajustar el código en un bucle, reduciendo así el coeficiente de, digamos,  $n^2$  o  $n^3$  en el tiempo de ejecución.

Deberíamos usar centinelas juiciosamente. Cuando hay muchas listas pequeñas, el almacenamiento adicional utilizado por sus centinelas puede representar un desperdicio de memoria significativo. En este libro, usamos centinelas solo cuando realmente simplifican el código.

### Ejercicios

#### 10.2-1

¿Puede implementar la operación de conjunto dinámico INSERTAR en una lista enlazada individualmente en tiempo  $O(1)$ ? ¿Qué hay de ELIMINAR?

#### 10.2-2

Implemente una pila usando una lista enlazada L. Las operaciones PUSH y POP aún deberían tomar  $O(1)$  tiempo.

#### 10.2-3

Implemente una cola mediante una lista enlazada L. Las operaciones ENQUEUE y DE QUEUE aún deberían tomar  $O(1)$  tiempo.

#### 10.2-4

Tal como está escrito, cada iteración de ciclo en el procedimiento LIST-SEARCH0 requiere dos pruebas: una para  $x \neq L:\text{nil}$  y otra para  $x:\text{key} \neq k$ . Muestre cómo eliminar la prueba para  $x \neq L:\text{nil}$  en cada iteración.

#### 10.2-5

Implemente las operaciones de diccionario INSERTAR, ELIMINAR y BUSCAR usando listas circulares de enlace simple. ¿Cuáles son los tiempos de ejecución de sus procedimientos?

## 10.2-6

La operación de conjuntos dinámicos UNION toma dos conjuntos disjuntos S1 y S2 como entrada y devuelve un conjunto SD S1 [ S2 que consta de todos los elementos de S1 y S2. Los conjuntos S1 y S2 suelen ser destruidos por la operación. Muestre cómo admitir UNION en tiempo O.1/ utilizando una estructura de datos de lista adecuada.

## 10.2-7

Proporcione un procedimiento no recursivo ,n/-time que invierta una lista enlazada de n elementos. El procedimiento no debe usar más que un almacenamiento constante más allá del necesario para la lista misma.

## 10.2-8 ?

Explique cómo implementar listas doblemente enlazadas utilizando solo un valor de puntero x:np por elemento en lugar de los dos habituales (siguiente y pre). Suponga que todos los valores de los punteros se pueden interpretar como enteros de k bits y defina x:np como x:np D x:next XOR x:pre, el k-bit "exclusive-or" de x:next y x:pre . (El valor NIL está representado por 0.)

Asegúrese de describir qué información necesita para acceder al encabezado de la lista. Muestre cómo implementar las operaciones de BÚSQUEDA, INSERCIÓN y ELIMINACIÓN en dicha lista.

Muestre también cómo invertir dicha lista en tiempo O.1/.

## 10.3 Implementando punteros y objetos

¿Cómo implementamos punteros y objetos en lenguajes que no los proporcionan?

En esta sección, veremos dos formas de implementar estructuras de datos enlazados sin un tipo de datos de puntero explícito. Sintetizaremos objetos y punteros a partir de matrices e índices de matrices.

### Una representación de matriz múltiple de objetos

Podemos representar una colección de objetos que tienen los mismos atributos usando una matriz para cada atributo. Como ejemplo, la figura 10.5 muestra cómo podemos implementar la lista enlazada de la figura 10.3(a) con tres arreglos. La clave de matriz contiene los valores de las claves actualmente en el conjunto dinámico y los punteros residen en las matrices next y pre. Para un índice de matriz dado x, las entradas de la matriz keyŒx, nextŒx y preŒx representan un objeto en la lista enlazada. Bajo esta interpretación, un puntero x es simplemente un índice común en las matrices key, next y pre .

En la Figura 10.3(a), el objeto con la clave 4 sigue al objeto con la clave 16 en la lista enlazada. En la figura 10.5, la tecla 4 aparece en la tecla 2, y la tecla 16 aparece en la tecla 5, y así sigue 5 D 2 y pre 2 D 5. Aunque la constante NIL aparece en la siguiente

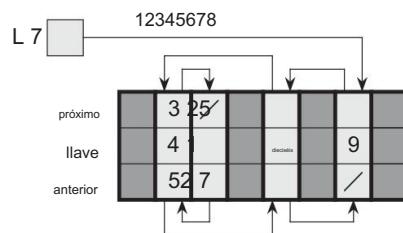


Figura 10.5 La lista enlazada de la Figura 10.3(a) representada por las matrices key, next y pre. Cada corte vertical de las matrices representa un solo objeto. Los punteros almacenados corresponden a los índices de matriz que se muestran en la parte superior; las flechas muestran cómo interpretarlos. Las posiciones de objetos ligeramente sombreadas contienen elementos de lista. La variable L guarda el índice de la cabeza.

atributo de la cola y el pre atributo de la cabeza, generalmente usamos un número entero (como 0 o 1) que no puede representar un índice real en las matrices. Una variable L contiene el índice de la cabeza de la lista.

#### Una representación de matriz única de objetos

Las palabras en la memoria de una computadora generalmente se direccionan con números enteros de 0 a M 1, donde M es un número entero adecuadamente grande. En muchos lenguajes de programación, un objeto ocupa un conjunto contiguo de ubicaciones en la memoria de la computadora. Un puntero es simplemente la dirección de la primera ubicación de memoria del objeto, y podemos dirigirnos a otras ubicaciones de memoria dentro del objeto agregando un desplazamiento al puntero.

Podemos usar la misma estrategia para implementar objetos en entornos de programación que no proporcionan tipos de datos de puntero explícitos. Por ejemplo, la figura 10.6 muestra cómo utilizar un solo arreglo A para almacenar la lista enlazada de las figuras 10.3(a) y 10.5. Un objeto ocupa un subarreglo contiguo  $A[i:j+k]$ . Cada atributo del objeto corresponde a un desplazamiento en el rango de 0 a  $k$ , y un puntero al objeto es el índice  $j$ .

En la figura 10.6, los desplazamientos correspondientes a key, next y pre son 0, 1 y 2, respectivamente. Para leer el valor de  $i:\text{pre}$ , dado un puntero  $i$ , sumamos el valor  $i$  del puntero al desplazamiento 2, leyendo así  $A[i+2]$ .

La representación de matriz única es flexible porque permite que objetos de diferentes longitudes se almacenen en la misma matriz. El problema de manejar una colección tan heterogénea de objetos es más difícil que el problema de manejar una colección homogénea, donde todos los objetos tienen los mismos atributos. Dado que la mayoría de las estructuras de datos que consideraremos están compuestas de elementos homogéneos, será suficiente para nuestros propósitos usar la representación de objetos en arreglos múltiples.

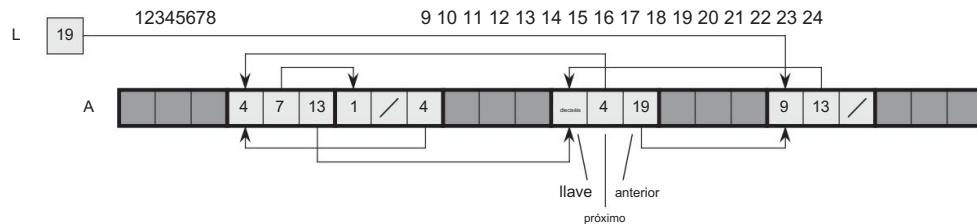


Figura 10.6 La lista enlazada de las Figuras 10.3(a) y 10.5 representada en un solo arreglo A. Cada elemento de la lista es un objeto que ocupa un subarreglo contiguo de longitud 3 dentro del arreglo. Los tres atributos key, next y pre corresponden a los desplazamientos 0, 1 y 2, respectivamente, dentro de cada objeto.

Un puntero a un objeto es el índice del primer elemento del objeto. Los objetos que contienen elementos de lista están ligeramente sombreados y las flechas muestran el orden de la lista.

### Asignar y liberar objetos

Para insertar una clave en un conjunto dinámico representado por una lista doblemente enlazada, debemos ubicar un puntero a un objeto actualmente no utilizado en la representación de la lista enlazada. Por lo tanto, es útil administrar el almacenamiento de objetos que no se utilizan actualmente en la representación de lista enlazada para que se pueda asignar uno. En algunos sistemas, un recolector de basura es responsable de determinar qué objetos no se utilizan. Muchas aplicaciones, sin embargo, son lo suficientemente simples como para asumir la responsabilidad de devolver un objeto no utilizado a un administrador de almacenamiento. Ahora exploraremos el problema de asignar y liberar (o desasignar) objetos homogéneos utilizando el ejemplo de una lista doblemente enlazada representada por múltiples arreglos.

Suponga que los arreglos en la representación de arreglos múltiples tienen una longitud m y que en algún momento el conjunto dinámico contiene nm elementos. Entonces n objetos representan elementos actualmente en el conjunto dinámico, y los mn objetos restantes son libres; los objetos libres están disponibles para representar elementos insertados en el conjunto dinámico en el futuro.

Mantenemos los objetos libres en una lista enlazada individualmente, a la que llamamos lista libre. La lista libre usa solo la siguiente matriz, que almacena los siguientes punteros dentro de la lista. La cabeza de la lista libre se mantiene en la variable global libre. Cuando el conjunto dinámico representado por la lista enlazada L no está vacío, la lista libre puede entrelazarse con la lista L, como se muestra en la figura 10.7. Tenga en cuenta que cada objeto de la representación está en la lista L o en la lista libre, pero no en ambas.

La lista libre actúa como una pila: el siguiente objeto asignado es el último liberado. Podemos usar una implementación de lista de las operaciones de pila PUSH y POP para implementar los procedimientos para asignar y liberar objetos, respectivamente. Suponemos que la variable global libre utilizada en los siguientes procedimientos apunta al primer elemento de la lista libre.

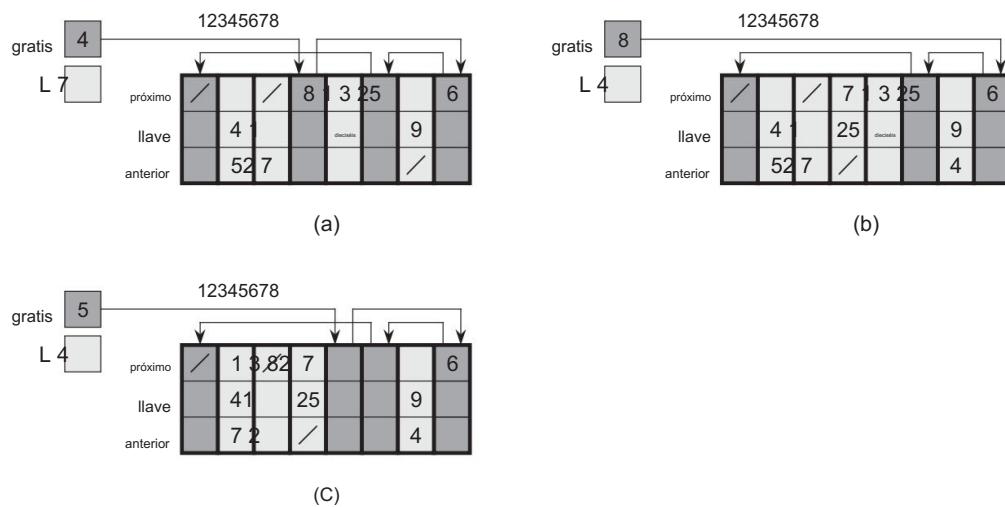


Figura 10.7 El efecto de los procedimientos ALLOCATE-OBJECT y FREE-OBJECT . (a) La lista de la figura 10.5 (ligeramente sombreada) y una lista libre (muy sombreada). Las flechas muestran la estructura de la lista libre. (b) El resultado de llamar a ALLOCATE-OBJECT./1 (que devuelve el índice 4), establecer keyŒ4 en 25 y llamar a LIST-INSERT.L; 4/. El nuevo encabezado de la lista libre es el objeto 8, que había sido el siguiente Œ4 en la lista libre. (c) Despues de ejecutar LIST-DELETE.L; 5/, lo llamamos OBJETO LIBRE.5/. El objeto 5 se convierte en el nuevo encabezado de la lista libre, con el objeto 8 siguiéndolo en la lista libre.

ALLOCATE-OBJECT./ 1 si

```

libre == NIL 2 error
“sin espacio” 3 si no x D libre 4
libre D x:siguiente
retorno x
5

```

FREE-OBJECT.x/ 1

```

x:siguiente D libre 2
libre D x

```

La lista libre contiene inicialmente todos los n objetos no asignados. Una vez que se ha agotado la lista libre, ejecutar el procedimiento ALLOCATE-OBJECT indica un error. Incluso podemos atender varias listas vinculadas con una sola lista gratuita. La figura 10.8 muestra dos listas enlazadas y una lista libre entrelazadas a través de matrices clave, siguiente y previa .

Los dos procedimientos se ejecutan en tiempo O.1/, lo que los hace bastante prácticos. Podemos modificarlos para que funcionen con cualquier colección homogénea de objetos dejando que cualquiera de los atributos del objeto actúe como el siguiente atributo en la lista libre.

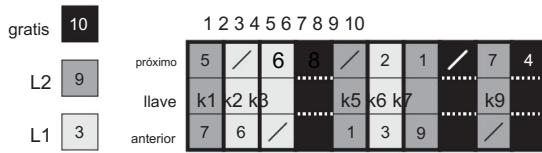


Figura 10.8 Dos listas enlazadas, L1 (ligeramente sombreada) y L2 (muy sombreada), y una lista libre (oscurecida) entrelazadas.

### Ejercicios

#### 10.3-1

Haz un dibujo de la secuencia h13; 4; 8; 19; 5; 11i almacenado como una lista doblemente enlazada utilizando la representación de matriz múltiple. Haga lo mismo para la representación de un solo arreglo.

#### 10.3-2

Escriba los procedimientos ALLOCATE-OBJECT y FREE-OBJECT para una colección homogénea de objetos implementados por la representación de matriz única.

#### 10.3-3

¿Por qué no necesitamos establecer o restablecer los atributos previos de los objetos en la implementación de los procedimientos ALLOCATE-OBJECT y FREE-OBJECT ?

#### 10.3-4

A menudo es deseable mantener compactos todos los elementos de una lista doblemente enlazada en el almacenamiento, utilizando, por ejemplo, las primeras ubicaciones de índice m en la representación de matriz múltiple. (Este es el caso en un entorno informático de memoria virtual paginado). Explique cómo implementar los procedimientos ALLOCATE-OBJECT y FREE-OBJECT para que la representación sea compacta. Suponga que no hay punteros a elementos de la lista enlazada fuera de la lista misma. (Sugerencia: use la implementación de matriz de una pila).

#### 10.3-5

Sea L una lista doblemente enlazada de longitud n almacenada en matrices key, pre y next de longitud m. Suponga que estas matrices son administradas por procedimientos ALLOCATE-OBJECT y FREE-OBJECT que mantienen una lista libre F doblemente enlazada. Suponga además que de los m elementos, exactamente n están en la lista L y mn están en la lista libre. Escriba un procedimiento COMPACTIFY-LIST.L; F / que, dada la lista L y la lista libre F, mueve los elementos, en L para que ocupen las posiciones 1 del arreglo; 2 : : : ; n y ajusta la lista libre F para que permanezca correcta, ocupando las posiciones de la matriz nC1; nC2;:::;m.

El tiempo de ejecución de su procedimiento debe ser ,n/, y debe usar solo una cantidad constante de espacio adicional. Argumenta que tu procedimiento es correcto.

## 10.4 Representación de árboles enraizados

Los métodos para representar listas dados en la sección anterior se extienden a cualquier estructura de datos homogénea. En esta sección, analizamos específicamente el problema de representar árboles con raíces mediante estructuras de datos enlazados. Primero observamos árboles binarios y luego presentamos un método para árboles con raíces en los que los nodos pueden tener un número arbitrario de hijos.

Representamos cada nodo de un árbol por un objeto. Al igual que con las listas enlazadas, asumimos que cada nodo contiene un atributo clave . Los restantes atributos de interés son punteros a otros nodos y varían según el tipo de árbol.

### árboles binarios

La figura 10.9 muestra cómo usamos los atributos p, izquierda y derecha para almacenar punteros al padre, al hijo izquierdo y al hijo derecho de cada nodo en un árbol binario T Si x:p D NIL, entonces x es la raíz. Si el nodo x no tiene hijo izquierdo, entonces x:left D NIL, y de manera similar para el hijo derecho. La raíz de todo el árbol T está señalada por el atributo T:raíz. Si T:root D NIL, entonces el árbol está vacío.

### Árboles enraizados con ramificación ilimitada.

Podemos extender el esquema para representar un árbol binario a cualquier clase de árboles en los que el número de hijos de cada nodo sea como máximo alguna constante k: reemplazamos los atributos izquierdo y derecho por hijo1; hijo2::::; niño Este esquema ya no funciona cuando el número de hijos de un nodo no está acotado, ya que no sabemos cuántos atributos (matrices en la representación de múltiples matrices) asignar por adelantado. Además, incluso si el número de hijos k está limitado por una constante grande pero la mayoría de los nodos tienen un número pequeño de hijos, podemos desperdiciar mucha memoria.

Afortunadamente, existe un esquema ingenioso para representar árboles con un número arbitrario de hijos. Tiene la ventaja de usar solo el espacio On/ para cualquier árbol con raíz de n nodos.

La representación del hijo izquierdo y el hermano derecho aparece en la figura 10.10. Como antes, cada nodo contiene un puntero padre p, y T:root apunta a la raíz del árbol T. Sin embargo, en lugar de tener un puntero a cada uno de sus hijos, cada nodo x tiene solo dos punteros:

1. x:left-child apunta al hijo más a la izquierda del nodo x, y
2. x:right-sibling apunta al hermano de x inmediatamente a su derecha.

Si el nodo x no tiene hijos, entonces x:hijo izquierdo D NIL, y si el nodo x es el hijo más a la derecha de su padre, entonces x:hermano derecho D NIL.

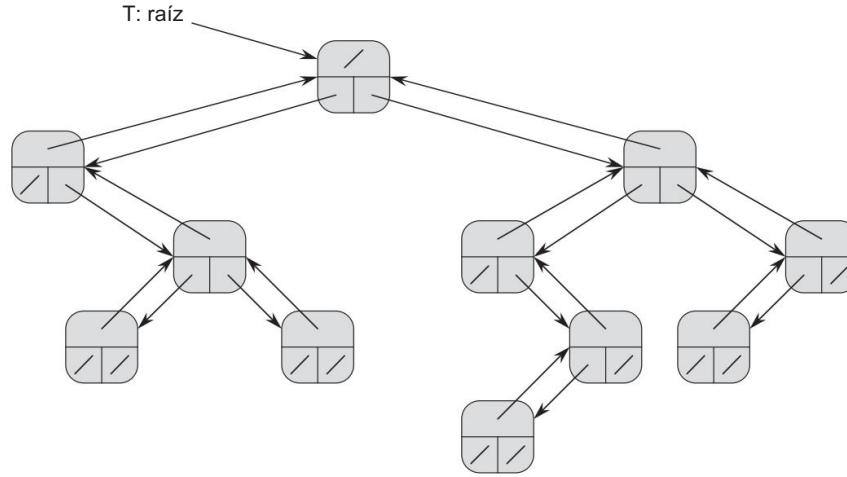


Figura 10.9 La representación de un árbol binario  $T$ . Cada nodo  $x$  tiene los atributos  $x:p$  (arriba),  $x:left$  (abajo a la izquierda) y  $x:right$  (abajo a la derecha). Los atributos clave no se muestran.

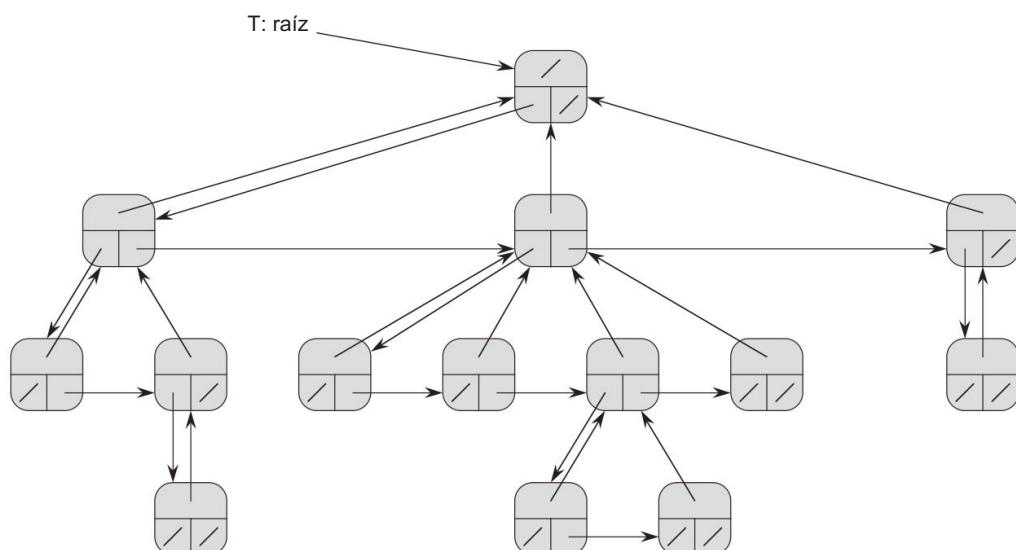


Figura 10.10 La representación del hijo izquierdo, hermano derecho de un árbol  $T$  (arriba). Cada nodo  $x$  tiene atributos  $x:p$  ( $x: \text{hijo izquierdo}$  (abajo a la izquierda) y  $x: \text{hermano derecho}$  (abajo a la derecha)). Los atributos clave no se muestran.

### Otras representaciones de árboles

A veces representamos árboles enraizados de otras maneras. En el Capítulo 6, por ejemplo, representamos un montón, que se basa en un árbol binario completo, mediante una única matriz más el índice del último nodo del montón. Los árboles que aparecen en el Capítulo 21 se recorren solo hacia la raíz, por lo que solo están presentes los punteros principales; no hay punteros a los niños. Muchos otros esquemas son posibles. Qué esquema es mejor depende de la aplicación.

### Ejercicios

#### 10.4-1

Dibuje el árbol binario con raíz en el índice 6 que está representado por los siguientes atributos:

clave de índice izquierda derecha			
1	12	7	3
2	15	8	NULO
3	4	10	NIL
4	10	5	9
5	2	NINGUNO	NINGUNO
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

#### 10.4-2

Escriba un procedimiento recursivo On-/time que, dado un árbol binario de n nodos, imprima la clave de cada nodo en el árbol.

#### 10.4-3

Escriba un procedimiento no recursivo On-/time que, dado un árbol binario de n nodos, imprima la clave de cada nodo en el árbol. Utilice una pila como estructura de datos auxiliar.

#### 10.4-4

Escriba un procedimiento On-/time que imprima todas las claves de un árbol con raíz arbitraria con n nodos, donde el árbol se almacene usando la representación de hijo izquierdo y hermano derecho.

#### 10.4-5 ?

Escriba un procedimiento no recursivo On-/time que, dado un árbol binario de n nodos, imprima la clave de cada nodo. No use más que espacio adicional constante afuera

del propio árbol y no modifique el árbol, ni siquiera temporalmente, durante el procedimiento.

#### 10.4-6 ?

La representación del hijo izquierdo y el hermano derecho de un árbol enraizado arbitrario utiliza tres punteros en cada nodo: hijo izquierdo, hermano derecho y padre. Desde cualquier nodo, su padre puede ser alcanzado e identificado en tiempo constante y todos sus hijos pueden ser alcanzados e identificados en tiempo lineal en el número de hijos. Mostrar cómo usar

solo dos punteros y un valor booleano en cada nodo para que el padre de un nodo o todos sus hijos puedan ser alcanzados e identificados en el tiempo lineal en el número de hijos.

## Problemas

### 10-1 Comparaciones entre listas Para

cada uno de los cuatro tipos de listas de la siguiente tabla, ¿cuál es el tiempo de ejecución asintótico en el peor de los casos para cada operación de conjunto dinámico enumerada?

	sin clasificar, enlazados individualmente	ordenados, enlazados individualmente	sin clasificar, dblemente enlazado	ordenada, dblemente enlazada
BUSCAR.L;				
k/				
INSERTAR.L;				
x/ ELIMINAR.L;				
x/ SUCESOR.L; x/				
ANTERIOR.L;				
x/ MINIMO.L/ MAXIMO.L/				

## 10-2 Montones combinables usando listas enlazadas

Un montón fusionable admite las siguientes operaciones: MAKE-HEAP (que crea un montón fusionable vacío), INSERT, MINIMUM, EXTRACT-MIN y UNION.<sup>1</sup>

Muestre cómo implementar montones combinables mediante listas vinculadas en cada uno de los siguientes casos.

Intente que cada operación sea lo más eficiente posible. Analice el tiempo de ejecución de cada operación en términos del tamaño de los conjuntos dinámicos en los que se opera.

a. Las listas están ordenadas.

b. Las listas están desordenadas.

C. Las listas no están ordenadas y los conjuntos dinámicos que se van a fusionar son inconexos.

## 10-3 Búsqueda en una lista compacta ordenada El

ejercicio 10.3-4 preguntaba cómo podríamos mantener una lista de n elementos de forma compacta en las primeras n posiciones de un arreglo. Supondremos que todas las claves son distintas y que la lista compacta también está ordenada, es decir, clave $\leq$ i < clave $\leq$ siguiente $\leq$ i para todo i D 1; 2; : : : ; n tal que next $\leq$ i  $\neq$  NIL. También supondremos que tenemos una variable L que contiene el índice del primer elemento de la lista. Bajo estas suposiciones, demostrará que podemos usar el siguiente algoritmo aleatorio para buscar en la lista en O.pn/tiempo esperado.

```
COMPACT-LIST-SEARCH.L; norte; k/
1 i DL 2
mientras i  $\neq$  NIL y key $\leq$ i < kj D RANDOM.1; n/ 3
    k           4 if key $\leq$ i < key $\leq$ j and key $\leq$ j
    5 i D j 6 if key $\leq$ i == k 7 return i 8 i D next $\leq$ i 9 if i == NIL or
        key $\leq$ i > k 10 return NIL 11
    else return i
```

Si ignoramos las líneas 3 a 7 del procedimiento, tenemos un algoritmo ordinario para buscar en una lista enlazada ordenada, en el que el índice i apunta a cada posición de la lista en

---

<sup>1</sup>Debido a que hemos definido un almacenamiento dinámico fusionable para admitir MINIMUM y EXTRACT-MIN, también podemos referirnos a él como un almacenamiento dinámico fusionable. Alternativamente, si fuera compatible con MAXIMUM y EXTRACT-MAX, sería un montón máximo fusionable.

doblar. La búsqueda termina una vez que el índice  $i$  "se cae" al final de la lista o una vez que  $\text{key}_i > k$ . En este último caso, si  $\text{key}_i = k$ , claramente hemos encontrado una clave con el valor  $k$ . Sin embargo, si  $\text{key}_i > k$ , entonces nunca encontraremos una clave con el valor  $k$ , por lo que terminar la búsqueda fue lo correcto.

Las líneas 3 a 7 intentan saltar a una posición  $j$  elegida al azar. Tal salto nos beneficia si  $\text{key}_j$  es mayor que  $\text{key}_i$  y no mayor que  $k$ ; en tal caso,  $j$  marca una posición en la lista que tendría que alcanzar durante una búsqueda de lista normal.

Debido a que la lista es compacta, sabemos que cualquier elección de  $j$  entre 1 y  $n$  indexa algún objeto en la lista en lugar de un espacio en la lista libre.

En lugar de analizar el rendimiento de COMPACT-LIST-SEARCH directamente, ejecutamos analizará un algoritmo relacionado, COMPACT-LIST-SEARCH0 bucles , dos separados. Este algoritmo toma un parámetro adicional  $t$  que determina un límite superior en el número de iteraciones del primer bucle.

```

COMPACT-LIST-SEARCH0 .L; norte; k; t/
1 i DL 2
para q D 1 a tj D
    RANDOM.1; n/ 3 si
    4 clave<math>\text{key}_i < \text{key}_j</math> y clave<math>\text{key}_j < k</math> j si clave<math>\text{key}_i =</math>
    5             = k
    6
        vuelvo yo
7 8 while i ≠ NIL and key<math>\text{key}_i < k</math> i
    next<math>\text{key}_i</math> 9 10
if i == NIL or key<math>\text{key}_i > k</math> 11 return
NIL
12 más volver i

```

Para comparar la ejecución de los algoritmos COMPACT-LIST-SEARCH.L; norte; k/ y COMPACT-LIST-SEARCH0 .L; norte; k; t/, suponga que la secuencia de enteros devuelta por las llamadas de RANDOM.1; n/ es el mismo para ambos algoritmos.

- Supongamos que COMPACT-LIST-SEARCH.L; norte; k/ toma  $t$  iteraciones del bucle while de las líneas 2-8. Argumente que COMPACT-LIST-SEARCH0 .L; norte; k; t/ devuelve la misma respuesta y que el número total de iteraciones de los bucles for y while dentro de COMPACT-LIST-SEARCH0 es al menos  $t$ .

En la llamada COMPACT-LIST-SEARCH0 .L; norte; k; t/, sea  $X_t$  la variable aleatoria que describe la distancia en la lista enlazada (es decir, a través de la cadena de punteros siguientes ) desde la posición  $i$  hasta la clave deseada  $k$  después de que hayan ocurrido iteraciones  $t$  del ciclo for de las líneas 2-7 .

b. Argumente que el tiempo de ejecución esperado de COMPACT-LIST-SEARCH0 .L; norte; k; t/ es  $Ot CE \Omega Ext /.$

C. Demuestre que  $E \Omega Ext Pn rD1.1 r=n/t$ . (Sugerencia: utilice la ecuación (C.25).)

d. Demuestre que  $P_{D0}^1 rt ntC1=.t C 1/.$

mi. Demostrar que  $E \Omega Ext n=.t C 1/.$

F. Muestre que COMPACT-LIST-SEARCH0 .L; norte; k; t/ corre en  $Ot C n=t/$  esperado tiempo.

gramo. Concluya que COMPACT-LIST-SEARCH se ejecuta en  $O.pn/\overline{t}$  tiempo esperado.

H. ¿Por qué asumimos que todas las claves son distintas en COMPACT-LIST-SEARCH?

Argumente que los saltos aleatorios no necesariamente ayudan asintóticamente cuando la lista contiene valores clave repetidos.

## Notas del capítulo

Aho, Hopcroft, Ullman [6] y Knuth [209] son excelentes referencias para estructuras de datos elementales. Muchos otros textos cubren tanto las estructuras de datos básicas como su implementación en un lenguaje de programación particular. Ejemplos de este tipo de libros de texto incluyen Goodrich y Tamassia [147], Main [241], Shaffer [311] y Weiss [352, 353, 354]. Gonnet [145] proporciona datos experimentales sobre el rendimiento de muchas operaciones de estructura de datos.

El origen de las pilas y colas como estructuras de datos en informática no está claro, ya que las nociones correspondientes ya existían en matemáticas y prácticas comerciales basadas en papel antes de la introducción de las computadoras digitales. Knuth [209] cita a AM Turing por el desarrollo de pilas para el enlace de subrutinas en 1947.

Las estructuras de datos basadas en punteros también parecen ser un invento popular. Según Knuth, aparentemente los punteros se usaban en las primeras computadoras con memorias de batería. El lenguaje A-1 desarrollado por GM Hopper en 1951 representaba fórmulas algebraicas como árboles binarios. Knuth le da crédito al lenguaje IPL-II, desarrollado en 1956 por A. Newell, JC Shaw y HA Simon, por reconocer la importancia y promover el uso de punteros. Su lenguaje IPL-III, desarrollado en 1957, incluía operaciones de pila explícitas.

---

## 11 tablas hash

Muchas aplicaciones requieren un conjunto dinámico que solo admite las operaciones de diccionario INSERTAR, BUSCAR y ELIMINAR. Por ejemplo, un compilador que traduce un lenguaje de programación mantiene una tabla de símbolos, en la que las claves de los elementos son cadenas de caracteres arbitrarias que corresponden a identificadores en el lenguaje. Una tabla hash es una estructura de datos efectiva para implementar diccionarios. Aunque la búsqueda de un elemento en una tabla hash puede llevar tanto tiempo como la búsqueda de un elemento en una lista enlazada ( $O(n)$  tiempo en el peor de los casos), en la práctica, el hash funciona extremadamente bien. Bajo suposiciones razonables, el tiempo promedio para buscar un elemento en una tabla hash es  $O(1)$ .

Una tabla hash generaliza la noción más simple de una matriz ordinaria. Dirigirse directamente a un arreglo ordinario hace un uso efectivo de nuestra habilidad para examinar una posición arbitraria en un arreglo en tiempo  $O(1)$ . La sección 11.1 analiza el direccionamiento directo con más detalle. Podemos aprovechar el direccionamiento directo cuando podemos permitirnos asignar una matriz que tenga una posición para cada tecla posible.

Cuando la cantidad de claves realmente almacenadas es pequeña en relación con la cantidad total de claves posibles, las tablas hash se convierten en una alternativa efectiva para dirigir directamente una matriz, ya que una tabla hash generalmente usa una matriz de tamaño proporcional a la cantidad de claves realmente almacenadas. En lugar de usar la clave como un índice de matriz directamente, el índice de matriz se calcula a partir de la clave. La Sección 11.2 presenta las ideas principales, centrándose en el "encadenamiento" como una forma de manejar las "colisiones", en las que más de una clave se asigna al mismo índice de matriz. La sección 11.3 describe cómo podemos calcular índices de matriz a partir de claves usando funciones hash. Presentamos y analizamos varias variaciones sobre el tema básico. La sección 11.4 analiza el "direcciónamiento abierto", que es otra forma de lidar con las colisiones. La conclusión es que el hashing es una técnica extremadamente efectiva y práctica: las operaciones básicas del diccionario requieren solo  $O(1)$  tiempo en promedio.

La Sección 11.5 explica cómo el "hashing perfecto" puede admitir búsquedas en  $O(1)$  tiempo en el peor de los casos, cuando el conjunto de claves que se almacena es estático (es decir, cuando el conjunto de claves nunca cambia una vez almacenado).

### 11.1 Tablas de direcciones directas

El direccionamiento directo es una técnica sencilla que funciona bien cuando el universo U de claves es razonablemente pequeño. Supongamos que una aplicación necesita un conjunto dinámico en el que cada elemento tenga una clave extraída del universo  $UD f0; 1; \dots; m 1g$ , donde  $m$  no es demasiado grande. Supondremos que no hay dos elementos que tengan la misma clave.

Para representar el conjunto dinámico, usamos una matriz, o tabla de direcciones directas, denotada por  $T \in 0 : : m 1$ , en la que cada posición, o ranura, corresponde a una clave en el universo U. La figura 11.1 ilustra el enfoque; la ranura k apunta a un elemento del conjunto con clave k. Si el conjunto no contiene ningún elemento con clave k, entonces  $T \in k D NIL$ .

Las operaciones del diccionario son triviales de implementar:

DIRECCIÓN DIRECTA-BÚSQUEDA.T; k / 1

retorno  $T \in k$

DIRECCIÓN DIRECTA-INSERTAR.T; x / 1

$T \in x : \text{tecla } D x$

DIRECCIÓN DIRECTA-BORRAR.T; x / 1 T

$\in x : \text{tecla } D NIL$

Cada una de estas operaciones requiere sólo  $O(1)$  de tiempo.

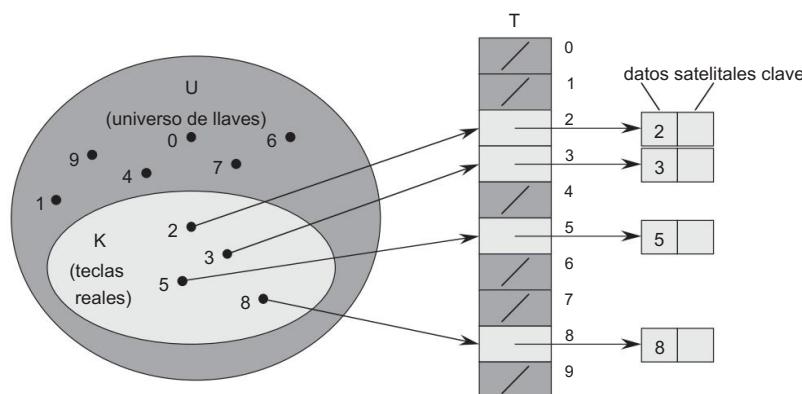


Figura 11.1 Cómo implementar un conjunto dinámico mediante una tabla de direcciones directas  $T \in UD f0; 1; \dots; 9g$ . Cada llave en el universo  $U$  determina una ranura en la tabla. Los conjuntos  $K \in f2; 3; 5; 8g$  de claves reales determinan las ranuras en la tabla que contienen punteros a elementos. Las otras ranuras, muy sombreadas, contienen NIL.

Para algunas aplicaciones, la propia tabla de direcciones directas puede contener los elementos del conjunto dinámico. Es decir, en lugar de almacenar la clave de un elemento y los datos del satélite en un objeto externo a la tabla de direcciones directas, con un puntero desde una ranura de la tabla al objeto, podemos almacenar el objeto en la ranura misma, ahorrando así espacio. Usaríamos una clave especial dentro de un objeto para indicar una ranura vacía. Además, muchas veces no es necesario almacenar la clave del objeto, ya que si tenemos el índice de un objeto en la tabla, tenemos su clave. Sin embargo, si las claves no están almacenadas, debemos tener alguna forma de saber si la ranura está vacía.

## Ejercicios

### 11.1-1

Suponga que un conjunto dinámico S está representado por una tabla de direcciones directas T de longitud m. Describa un procedimiento que encuentre el elemento máximo de S. ¿Cuál es el desempeño de su procedimiento en el peor de los casos?

### 11.1-2

Un vector de bits es simplemente una matriz de bits (0 y 1). Un vector de bits de longitud m ocupa mucho menos espacio que una matriz de m punteros. Describir cómo usar un vector de bits para representar un conjunto dinámico de elementos distintos sin datos de satélite. Las operaciones de diccionario deben ejecutarse en tiempo O.1/.

### 11.1-3

Sugiera cómo implementar una tabla de direcciones directas en la que las claves de los elementos almacenados no necesiten ser distintas y los elementos puedan tener datos satelitales. Todo tres operaciones de diccionario (INSERTAR, ELIMINAR y BUSCAR) deben ejecutarse en tiempo O.1/. (No olvide que DELETE toma como argumento un puntero a un objeto que se eliminará, no una clave).

### 11.1-4 ?

Deseamos implementar un diccionario usando direccionamiento directo en una gran matriz. Al principio, las entradas de la matriz pueden contener basura, y la inicialización de toda la matriz no es práctica debido a su tamaño. Describa un esquema para implementar un diccionario de direcciones directas en una gran matriz. Cada objeto almacenado debe usar el espacio O.1/; las operaciones BUSCAR, INSERTAR y ELIMINAR deben tomar O.1/ tiempo cada una; e inicializar la estructura de datos debería tomar O.1/ tiempo. (Sugerencia: use una matriz adicional, tratada como una pila cuyo tamaño es el número de claves realmente almacenadas en el diccionario, para ayudar a determinar si una entrada dada en la enorme matriz es válida o no).

---

## 11.2 Tablas hash

La desventaja del direccionamiento directo es obvia: si el universo  $U$  es grande, almacenar una tabla  $T$  de tamaño  $|U|$  puede ser poco práctico, o incluso imposible, dada la memoria disponible en una computadora típica. Además, el conjunto  $K$  de claves almacenadas en realidad puede ser tan pequeño en relación con  $U$  que la mayor parte del espacio asignado para  $T$  se desperdiciaría.

Cuando el conjunto  $K$  de claves almacenadas en un diccionario es mucho más pequeño que el universo  $U$  de todas las claves posibles, una tabla hash requiere mucho menos almacenamiento que una tabla de direcciones directas. Específicamente, podemos reducir el requisito de almacenamiento a  $|K|$ , mientras mantenemos el beneficio de que la búsqueda de un elemento en la tabla hash requiere solo  $O(1)$  de tiempo. El problema es que este límite es para el tiempo de caso promedio, mientras que para el direccionamiento directo se mantiene para el tiempo de caso más desfavorable.

Con direccionamiento directo, un elemento con clave  $k$  se almacena en la ranura  $k$ . Con hashing, este elemento se almacena en la ranura  $h(k)$ ; es decir, usamos una función hash  $h$  para calcular la ranura de la clave  $k$ . Aquí,  $h$  mapea el universo  $U$  de claves en las ranuras de una tabla hash  $T : : m$ :

$h : U \rightarrow \{0, 1, \dots, m-1\}$

donde el tamaño  $m$  de la tabla hash suele ser mucho menor que  $|U|$ . Decimos que un elemento con clave  $k$  genera un hash en la ranura  $h(k)$ ; también decimos que  $h(k)$  es el valor hash de la clave  $k$ . La figura 11.2 ilustra la idea básica. La función hash reduce el rango de índices de matriz y, por lo tanto, el tamaño de la matriz. En lugar de un tamaño de  $|U|$ , la matriz puede tener un tamaño  $m$ .

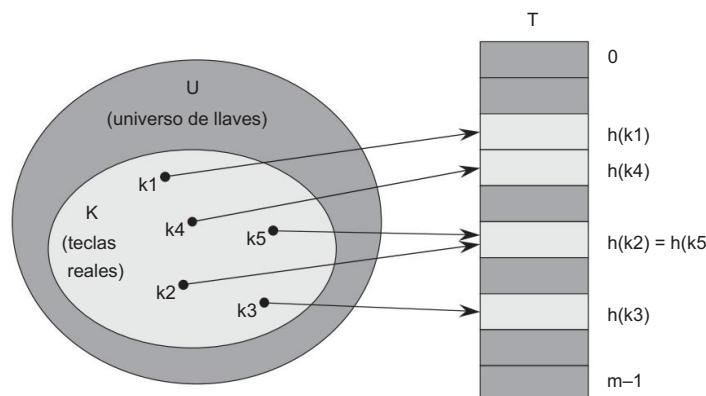


Figura 11.2 Uso de una función hash  $h$  para asignar claves a ranuras de tablas hash. Debido a que las teclas  $k_2$  y  $k_5$  se asignan a la misma ranura, chocan.

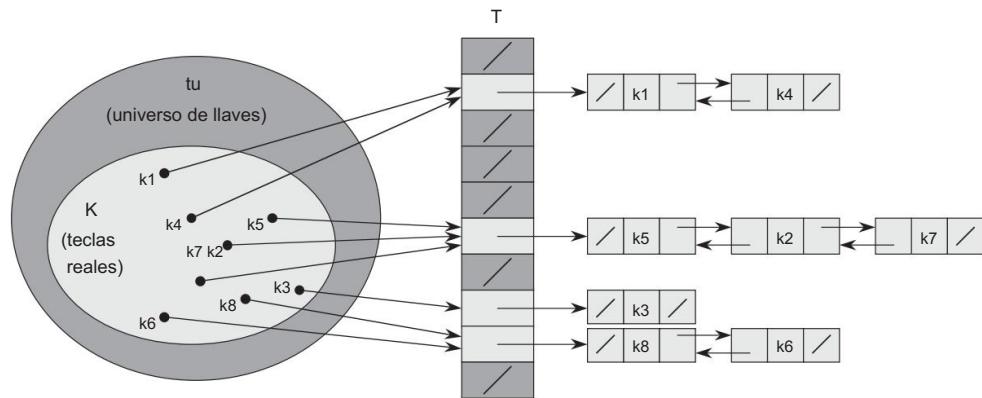


Figura 11.3 Resolución de colisiones por encadenamiento. Cada ranura de tabla hash T  $\in j$  contiene una lista enlazada de todas las claves cuyo valor hash es  $j$ . Por ejemplo,  $h.k1 \in D h.k4 \wedge h.k5 \in D h.k7 \wedge h.k2 \in D h.k6$ . La lista enlazada puede estar enlazada simple o doblemente; lo mostramos como doblemente vinculado porque la eliminación es más rápida de esa manera.

Hay un problema: dos claves pueden codificarse en la misma ranura. A esta situación la llamamos colisión. Afortunadamente, contamos con técnicas efectivas para resolver el conflicto creado por las colisiones.

Por supuesto, la solución ideal sería evitar las colisiones por completo. Podríamos tratar de lograr este objetivo eligiendo una función hash  $h$  adecuada. Una idea es hacer que  $h$  parezca "aleatorio", evitando así las colisiones o al menos minimizando su número. El mismo término "hacer hash", que evoca imágenes de mezclas y cortes aleatorios, capture el espíritu de este enfoque. (Por supuesto, una función hash  $h$  debe ser determinista en el sentido de que una entrada dada  $k$  siempre debe producir la misma salida  $hk$ ).

Sin embargo, debido a que  $jUj > m$ , debe haber al menos dos claves que tengan el mismo valor hash; por lo tanto, evitar las colisiones por completo es imposible. Por lo tanto, si bien una función hash de aspecto "aleatorio" bien diseñada puede minimizar el número de colisiones, todavía necesitamos un método para resolver las colisiones que ocurren.

El resto de esta sección presenta la técnica de resolución de colisiones más simple, llamada encadenamiento. La Sección 11.4 introduce un método alternativo para resolver colisiones, llamado direccionamiento abierto.

#### Resolución de colisiones por encadenamiento

En el encadenamiento, colocamos todos los elementos que hacen hash en el mismo espacio en la misma lista enlazada, como muestra la figura 11.3. La ranura  $j$  contiene un puntero al encabezado de la lista de todos los elementos almacenados cuyo hash es  $j$ ; si no existen tales elementos, la ranura  $j$  contiene NIL.

Las operaciones de diccionario en una tabla hash T son fáciles de implementar cuando las colisiones se resuelven encadenando:

ENCADENADO-HASH-INSERT.T; x/ 1

insertar x al principio de la lista T  $\langle\!\!> h.x:\text{key}$

ENCADENADO-HASH-SEARCH.T; k/ 1

busca un elemento con clave k en la lista T  $\langle\!\!> h.k$

ENCADENADO-HASH-DELETE.T; x/ 1

borrar x de la lista T  $\langle\!\!> h.x:\text{key}$

El peor tiempo de ejecución para la inserción es O.1/. El procedimiento de inserción es rápido en parte porque asume que el elemento x que se está insertando no está ya presente en la tabla; si es necesario, podemos verificar esta suposición (con un costo adicional) buscando un elemento cuya clave sea x:key antes de insertarlo. Para la búsqueda, el tiempo de ejecución del peor de los casos es proporcional a la longitud de la lista; analizaremos más de cerca esta operación a continuación. Podemos eliminar un elemento en O.1/ tiempo si las listas están doblemente vinculadas, como muestra la figura 11.3. (Tenga en cuenta que CHAINED-HASH-DELETE toma como entrada un elemento x y no su clave k, por lo que no tenemos que buscar x primero. Si la tabla hash admite la eliminación, entonces sus listas enlazadas deben estar doblemente enlazadas para que podemos eliminar un elemento rápidamente. Si las listas solo estuvieran vinculadas individualmente, entonces para eliminar el elemento x, primero tendríamos que encontrar x en la lista T  $\langle\!\!> h.x:\text{key}$  para que podamos actualizar el siguiente atributo del predecesor de x. Con listas enlazadas individualmente, tanto la eliminación como la búsqueda tendrían los mismos tiempos de ejecución asintóticos).

#### Análisis de hash con encadenamiento

¿Qué tan bien funciona el hashing con encadenamiento? En particular, ¿cuánto tiempo lleva buscar un elemento con una clave determinada?

Dada una tabla hash T con m ranuras que almacena n elementos, definimos el factor de carga  $\lambda$  para T como  $n=m$ , es decir, el número promedio de elementos almacenados en una cadena.

Nuestro análisis será en términos de  $\lambda$ , que puede ser menor, igual o mayor que 1.

El comportamiento en el peor de los casos de hashing con encadenamiento es terrible: todas las n claves se codifican en la misma ranura, creando una lista de longitud n. El tiempo de búsqueda en el peor de los casos es, por lo tanto,  $\lambda \cdot n$  más el tiempo para calcular la función hash, no mejor que si usáramos una lista enlazada para todos los elementos. Claramente, no usamos tablas hash para su desempeño en el peor de los casos. (Sin embargo, el hashing perfecto, descrito en la Sección 11.5, proporciona un buen rendimiento en el peor de los casos cuando el conjunto de claves es estático).

El rendimiento de hash de caso promedio depende de qué tan bien la función hash h distribuye el conjunto de claves que se almacenarán entre las m ranuras, en promedio.

La sección 11.3 analiza estos problemas, pero por ahora supondremos que cualquier elemento dado tiene la misma probabilidad de generar hash en cualquiera de las  $m$  ranuras, independientemente de dónde se haya generado hash para cualquier otro elemento. Llamamos a esto la

Conjunto de suposición de hashing uniforme simple. Si  $m = 1$ , denotemos la longitud de la lista  $T$  de modo que  
 $n = D \leq C n_1 \leq m n_1$ ; (11.1)

y el valor esperado de  $n_j$  es  $E[n_j] = D$ ,  $D = m$ .

Suponemos que el tiempo  $O(1)$  es suficiente para calcular el valor hash  $h_k$ , por lo que el tiempo requerido para buscar un elemento con clave  $k$  depende linealmente de la longitud  $n h_k$  de la lista  $T \in h_k$ . Dejando de lado el tiempo  $O(1)$  requerido para calcular la función hash y acceder a la ranura  $h_k$ , consideremos el número esperado de elementos examinados por el algoritmo de búsqueda, es decir, el número de elementos en la lista  $T \in h_k$  que el algoritmo comprueba si alguno tiene una clave igual a  $k$ . Consideraremos dos casos. En el primero, la búsqueda no tiene éxito: ningún elemento de la tabla tiene la clave  $k$ . En el segundo, la búsqueda encuentra con éxito un elemento con clave  $k$ .

#### Teorema 11.1

En una tabla hash en la que las colisiones se resuelven mediante el encadenamiento, una búsqueda fallida requiere un tiempo de caso promedio de  $\sqrt{C}$ , bajo el supuesto de hash uniforme simple.

Prueba Bajo el supuesto de hashing uniforme simple, es igualmente probable que cualquier clave  $k$  que no esté almacenada en la tabla se convierta en hash en cualquiera de las  $m$  ranuras. El tiempo esperado para buscar sin éxito una clave  $k$  es el tiempo esperado para buscar hasta el final de la lista  $T \in h_k$ , que tiene una longitud esperada  $E[n_h_k] = D$ . Por lo tanto, el número esperado de elementos examinados en una búsqueda fallida es  $\sqrt{D}$ , y el tiempo total requerido (incluido el tiempo para calcular  $h_k$ ) es  $\sqrt{C}$ . ■

La situación para una búsqueda exitosa es ligeramente diferente, ya que no es igualmente probable que se busque en cada lista. En cambio, la probabilidad de que se busque en una lista es proporcional al número de elementos que contiene. No obstante, el tiempo de búsqueda esperado todavía resulta ser  $\sqrt{C}$ .

#### Teorema 11.2

En una tabla hash en la que las colisiones se resuelven mediante el encadenamiento, una búsqueda exitosa requiere un tiempo de caso promedio  $\sqrt{C}$ , bajo el supuesto de hash uniforme simple.

Prueba Suponemos que el elemento que se busca es igualmente probable que sea cualquiera de los  $n$  elementos almacenados en la tabla. El número de elementos examinados durante una búsqueda exitosa de un elemento  $x$  es uno más que el número de elementos que

aparecer antes de  $x$  en la lista de  $x$ . Debido a que los elementos nuevos se colocan al principio de la lista, los elementos antes de  $x$  en la lista se insertaron después de que se insertó  $x$ . Para encontrar el número esperado de elementos examinados, tomamos el promedio, sobre los  $n$  elementos  $x$  en la tabla, de 1 más el número esperado de elementos agregados a la lista de  $x$  después de que  $x$  se agregó a la lista. Sea  $x_i$  el  $i$ -ésimo elemento insertado en la tabla, para  $i \in \{1; 2; \dots; n\}$ , y sea  $k_i \in D$  :key. Para las claves definimos el  $k_i$  y  $k_j$ , variable aleatoria indicadora  $X_{ij} = 1$  si  $k_i = k_j$ . Bajo el supuesto de hashing uniforme simple, tenemos  $\Pr[k_i = k_j] = 1/m$ , y así por el Lema 5.1,  $E[X_{ij}] = 1/m$ . Por lo tanto, el número esperado de elementos examinados en un buscar es

$$\begin{aligned}
& \text{mi } "1" \stackrel{\text{D}}{\sim} \frac{1}{m} X_n \stackrel{iD1}{\sim} \frac{1}{m} C X_n \stackrel{jDiC1}{\sim} \frac{1}{m} X_{ij} \# \\
& \text{D} \quad \frac{1}{m} X_n \stackrel{iD1}{\sim} \frac{1}{m} C X_n \stackrel{jDiC1}{\sim} E C X_{ij} ! \text{ (por linealidad de la expectativa)} \\
& \text{D} \quad \frac{1}{m} X_n \stackrel{iD1}{\sim} \frac{1}{m} C X_n \stackrel{jDiC1}{\sim} \frac{1}{m} ! \\
& \text{D } 1 C \quad \frac{1}{m} X_n \stackrel{iD1}{\sim} .ni / \\
& \text{Nuevo Méjico} \\
& \text{D } 1 C \quad \frac{1}{m} X_n \stackrel{iD1}{\sim} x_n x_n \stackrel{j!}{\sim} \\
& \text{D } 1 C \quad \frac{1}{m} n^2 \quad \frac{n n C 1/2}{2m} \quad \text{(por ecuación (A.1))} \\
& \text{D } 1 C \quad \frac{n^2}{2m} \\
& \text{D } 1 C 2 \quad \frac{n^2}{2m}
\end{aligned}$$

Por lo tanto, el tiempo total requerido para una búsqueda exitosa (incluyendo el tiempo para calcular la función hash) es  $\dots C_2 = 2n/ D \dots C_1$ .

¿Qué significa este análisis? Si el número de espacios en la tabla hash es al menos proporcional al número de elementos en la tabla, tenemos  $n \propto D \cdot m$  y, en consecuencia,  $D \propto n/m = m/D$ . Por lo tanto, la búsqueda lleva un tiempo constante en promedio. Dado que la inserción toma  $O(1)$  en el peor de los casos y la eliminación toma  $O(1)$  en el peor de los casos cuando las listas están doblemente vinculadas, podemos admitir todas las operaciones de diccionario en  $O(1)$  en promedio.

### Ejercicios

#### 11.2-1

Supongamos que usamos una función hash  $h$  para hacer hash de  $n$  claves distintas en un arreglo  $T$  de longitud  $m$ . Suponiendo un hashing uniforme simple, ¿cuál es el número esperado de colisiones? Más precisamente, ¿cuál es la cardinalidad esperada de  $\sum_{k=1}^m \sum_{l \neq k} I_{\{h(k) = h(l)\}}$ ?

#### 11.2-2

Demostrar lo que sucede cuando insertamos las teclas 5; 28; 19; 15; 20; 33; 12; 17; 10 en una tabla hash con colisiones resueltas por encadenamiento. Deje que la tabla tenga 9 ranuras y deje que la función hash sea  $h(k) = k \bmod 9$ .

#### 11.2-3

El profesor Marley plantea la hipótesis de que puede obtener mejoras sustanciales en el rendimiento modificando el esquema de encadenamiento para mantener cada lista ordenada. ¿Cómo afecta la modificación del profesor al tiempo de ejecución de búsquedas exitosas, búsquedas fallidas, inserciones y eliminaciones?

#### 11.2-4

Sugiera cómo asignar y desasignar almacenamiento para elementos dentro de la propia tabla hash vinculando todos los espacios no utilizados en una lista libre. Suponga que una ranura puede almacenar una bandera y un elemento más un puntero o dos punteros. Todas las operaciones de diccionario y lista libre deben ejecutarse en el tiempo esperado  $O(1)$ . ¿Es necesario que la lista libre esté doblemente enlazada o basta con una lista libre enlazada individualmente?

#### 11.2-5

Supongamos que estamos almacenando un conjunto de  $n$  claves en una tabla hash de tamaño  $m$ . Muestre que si las claves se extraen de un universo  $U$  con  $|U| > nm$ , entonces  $U$  tiene un subconjunto de tamaño  $n$  que consta de claves que todas se codifican en la misma ranura, de modo que el tiempo de búsqueda en el peor de los casos para el cifrado con encadenamiento es .. norte/.

#### 11.2-6

Supongamos que hemos almacenado  $n$  claves en una tabla hash de tamaño  $m$ , con colisiones resueltas por encadenamiento, y que conocemos la longitud de cada cadena, incluida la longitud  $L$  de la cadena más larga. Describa un procedimiento que seleccione una clave uniformemente al azar de entre las claves de la tabla hash y la devuelva en el tiempo esperado  $O(1)$ .

## 11.3 Funciones hash

En esta sección, discutimos algunos temas relacionados con el diseño de buenas funciones hash y luego presentamos tres esquemas para su creación. Dos de los esquemas, hashing por división y hashing por multiplicación, son de naturaleza heurística, mientras que el tercer esquema, hashing universal, utiliza la aleatorización para proporcionar un rendimiento demostrablemente bueno.

¿Qué hace que una buena función hash?

Una buena función hash satisface (aproximadamente) la suposición de hash uniforme simple: cada clave tiene la misma probabilidad de generar hash en cualquiera de las  $m$  ranuras, independientemente de dónde se haya generado cualquier otra clave. Desafortunadamente, normalmente no tenemos forma de verificar esta condición, ya que rara vez conocemos la distribución de probabilidad de la que se extraen las claves. Además, es posible que las claves no se dibujen de forma independiente.

De vez en cuando conocemos la distribución. Por ejemplo, si sabemos que las claves son números reales aleatorios  $k$  distribuidos de manera independiente y uniforme en el rango  $0 \leq k < 1$ , entonces la función hash

$h(k) = D \text{ negro}$

satisface la condición de hashing uniforme simple.

En la práctica, a menudo podemos emplear técnicas heurísticas para crear una función hash que funcione bien. La información cualitativa sobre la distribución de claves puede ser útil en este proceso de diseño. Por ejemplo, considere la tabla de símbolos de un compilador, en la que las claves son cadenas de caracteres que representan identificadores en un programa. Los símbolos estrechamente relacionados, como `pt` y `pts`, a menudo aparecen en el mismo programa. Una buena función de hash minimizaría la posibilidad de que dichas variantes se hayan utilizado en el mismo espacio.

Un buen enfoque deriva el valor hash de manera que esperamos que sea independiente de cualquier patrón que pueda existir en los datos. Por ejemplo, el "método de división" (discutido en la Sección 11.3.1) calcula el valor hash como el resto cuando la clave se divide por un número primo específico. Este método frecuentemente da buenos resultados, asumiendo que elegimos un número primo que no está relacionado con ningún patrón en la distribución de claves.

Finalmente, notamos que algunas aplicaciones de funciones hash pueden requerir propiedades más fuertes que las proporcionadas por un hash uniforme simple. Por ejemplo, es posible que queramos claves que sean "cercanas" en algún sentido para producir valores hash que estén muy separados. (Esta propiedad es especialmente deseable cuando usamos el sondeo lineal, definido en la Sección 11.4.) El hashing universal, descrito en la Sección 11.3.3, a menudo proporciona las propiedades deseadas.

### Interpretar claves como números naturales

La mayoría de las funciones hash asumen que el universo de claves es el conjunto  $\{0, 1, 2, \dots, g\}$  de números naturales. Así, si las claves no son números naturales, encontramos la forma de interpretarlas como números naturales. Por ejemplo, podemos interpretar una cadena de caracteres como un número entero expresado en notación de base adecuada. Así, podríamos interpretar el identificador  $pt$  como el par de enteros decimales  $(112, 116)$ , ya que  $p \in \{0, 1, \dots, 127\}$  y  $t \in \{0, 1, \dots, 127\}$  en el conjunto de caracteres ASCII; luego, expresado como un entero de base 128,  $pt$  se convierte en  $112 \cdot 128 + 116 = 14452$ . En el contexto de una aplicación dada, generalmente podemos idear algún método para interpretar cada clave como un número natural (posiblemente grande). En lo que sigue, asumimos que las claves son números naturales.

#### 11.3.1 El método de división

En el método de división para crear funciones hash, mapeamos una clave  $k$  en una de  $m$  ranuras tomando el resto de  $k$  dividido por  $m$ . Es decir, la función hash es

$$h_k = k \bmod m$$

Por ejemplo, si la tabla hash tiene un tamaño  $m = 12$  y la clave es  $k = 100$ , entonces  $h_k = 4$ . Dado que solo requiere una sola operación de división, el hash por división es bastante rápido.

Cuando usamos el método de división, generalmente evitamos ciertos valores de  $m$ . Por ejemplo,  $m$  no debería ser una potencia de 2, ya que si  $m = 2^p$ , entonces  $h_k$  son solo los  $p$  bits de orden más bajo de  $k$ . A menos que sepamos que todos los patrones de bits  $p$  de orden bajo son igualmente probables, es mejor diseñar la función hash para que dependa de todos los bits de la clave. Como le pide que muestre el ejercicio 11.3-3, elegir  $m = 2^p - 1$  cuando  $k$  es una cadena de caracteres interpretada en base  $2^p$  puede ser una mala elección, porque permutar los caracteres de  $k$  no cambia su valor hash.

Un número primo que no se acerque demasiado a una potencia exacta de 2 suele ser una buena elección para  $m$ . Por ejemplo, supongamos que deseamos asignar una tabla hash, con colisiones resueltas por encadenamiento, para contener aproximadamente  $n = 2000$  cadenas de caracteres, donde un carácter tiene 8 bits. No nos importa examinar un promedio de 3 elementos en una búsqueda fallida, por lo que asignamos una tabla hash de tamaño  $m = 701$ . Podríamos elegir  $m = 701$  porque es un número primo cercano a  $2000/3$  pero no cercano a ninguna potencia de 2. Al tratar cada clave  $k$  como un número entero, nuestra función

$$\text{hash} = h_k = k \bmod 701$$

#### 11.3.2 El método de multiplicación

El método de multiplicación para crear funciones hash opera en dos pasos. Primero, multiplicamos la clave  $k$  por una constante  $A$  en el rango  $0 < A < 1$  y extraemos la

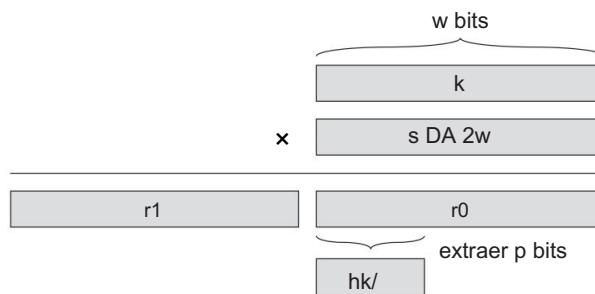


Figura 11.4 El método de multiplicación de hashing. La representación en bits  $w$  de la clave  $k$  se multiplica por el valor en bits  $w$   $s$  DA  $2w$ . Los  $p$  bits de mayor orden de la mitad inferior del bit  $w$  del producto forman el valor hash deseado  $hk/$ .

parte fraccionaria de  $kA$ . Luego, multiplicamos este valor por  $m$  y tomamos el piso del resultado. En resumen, la función hash es

$$hk/ \leftarrow D \lfloor b^m .kA \bmod 1 \right\rfloor c ;$$

donde “ $kA \bmod 1$ ” significa la parte fraccionaria de  $kA$ , es decir,  $kA - bkA$ .

Una ventaja del método de multiplicación es que el valor de  $m$  no es crítico.

Por lo general, elegimos que sea una potencia de 2 ( $m = 2^p$  para algún número entero  $p$ ), ya que podemos implementar fácilmente la función en la mayoría de las computadoras de la siguiente manera. Suponga que el tamaño de palabra de la máquina es de  $w$  bits y que  $k$  cabe en una sola palabra. Restringimos  $A$  para que sea una fracción de la forma  $s=2^w$ , donde  $s$  es un número entero en el rango  $0 < s < 2^w$ . Haciendo referencia a la figura 11.4, primero multiplicamos  $k$  por el número entero del bit  $w$   $s$  DA  $2w$ . El resultado es un valor de  $2w$  bits  $r_1r_2\dots r_0$ , donde  $r_1$  es la palabra de orden superior del producto y  $r_0$  es la palabra de orden inferior del producto. El valor hash de  $p$  bits deseado consta de los  $p$  bits más significativos de  $r_0$ .

Aunque este método funciona con cualquier valor de la constante  $A$ , funciona mejor con algunos valores que con otros. La elección óptima depende de las características de los datos que se procesan. Knuth [211] sugiere que

$$A = \frac{1}{2} \quad m = 6180339887 \quad \dots \quad (11.2)$$

es probable que funcione razonablemente bien.

Como ejemplo, supongamos que tenemos  $k = 123456$ ,  $p = 14$ ,  $m = 214$  y  $w = 32$ . Adaptando la sugerencia de Knuth, elegimos  $A$  para que sea la fracción de la forma  $s=232$  más cercana a  $\frac{1}{2}$ , por lo que  $A = \frac{2654435769}{2^{32}} = 232$ . Entonces  $ks = 327706022297664$  y  $232/C = 17612864$ , y así  $r_1 = 76300$  y  $r_0 = 17612864$ . Los 14 bits más significativos de  $r_0$  dan el valor  $hk/ = 67$ .

### 11.3.3 Hash universal

Si un adversario malicioso elige las claves que se van a codificar mediante alguna función de hash fija, entonces el adversario puede elegir  $n$  claves que todas se conviertan en hash en la misma ranura, lo que arroja un tiempo de recuperación promedio de  $\sqrt{n}$ . Cualquier función hash fija es vulnerable a este terrible comportamiento en el peor de los casos; la única forma efectiva de mejorar la situación es elegir la función hash al azar de una manera que sea independiente de las claves que realmente se van a almacenar. Este enfoque, llamado hashing universal, puede generar un rendimiento promedio demostrablemente bueno, sin importar qué claves elija el adversario.

En el hashing universal, al comienzo de la ejecución seleccionamos la función hash al azar de una clase de funciones cuidadosamente diseñadas. Al igual que en el caso de la ordenación rápida, la aleatorización garantiza que ninguna entrada única siempre evocará el peor de los casos. Debido a que seleccionamos aleatoriamente la función hash, el algoritmo se puede tener de manera diferente en cada ejecución, incluso para la misma entrada, lo que garantiza un buen rendimiento de caso promedio para cualquier entrada. Volviendo al ejemplo de la tabla de símbolos de un compilador, encontramos que la elección de identificadores por parte del programador ahora no puede causar un rendimiento de hash consistentemente bajo. El rendimiento deficiente ocurre solo cuando el compilador elige una función hash aleatoria que hace que el conjunto de identificadores tenga un hash deficiente, pero la probabilidad de que ocurra esta situación es pequeña y es la misma para cualquier conjunto de identificadores del mismo tamaño.

Sea  $H$  una colección finita de funciones hash que mapean un universo dado  $U$  de claves en el rango  $f_0, f_1, \dots, f_{m-1}$ . Se dice que tal colección es universal si para cada par de claves distintas  $k, l \in U$ , el número de funciones hash  $h \in H$  para las cuales  $h(k) = h(l)$  es como mucho  $\frac{1}{m}$ . En otras palabras, con una función hash elegida al azar de  $H$ , la probabilidad de una colisión entre las distintas claves  $k$  y  $l$  no es mayor que la probabilidad  $1/m$  de una colisión si  $h(k) = h(l)$  se eligieran aleatoriamente del conjunto  $f_0, f_1, \dots, f_{m-1}$ .

El siguiente teorema muestra que una clase universal de funciones hash da buena comportamiento de caso promedio. Recuerde que  $|T|$  denota la longitud de la lista  $T$ .

#### Teorema 11.3

Suponga que una función hash  $h$  se elige aleatoriamente de una colección universal de funciones hash y se ha utilizado para codificar  $n$  claves en una tabla  $T$  de tamaño  $m$ , usando encadenamiento para resolver colisiones. Si la clave  $k$  no está en la tabla, entonces la longitud esperada  $E[|h(k)|]$  de la lista a la que la clave  $k$  pertenece es como máximo el factor de carga  $C$ ,  $D = n/m$ .

Si la clave  $k$  está en la tabla, entonces la longitud esperada  $E[|h(k)|]$  de la lista que contiene la clave  $k$  es como máximo  $1 + C$ .

Prueba Notamos que las expectativas aquí están sobre la elección de la función hash y no dependen de ninguna suposición sobre la distribución de las claves.

Para cada par  $k, l$  de claves distintas, defina la variable aleatoria indicadora

$X_{kl} \in f_h(k) / D$   $h_l/g$ . Dado que por definición de una colección universal de funciones hash, un solo par de claves choca con una probabilidad máxima de  $1=m$ , tenemos  $\Pr[f_h(k) / D = h_l/g] = m$ . Por el Lema 5.1, por lo tanto, tenemos  $E[\#X_{kl}] = m$ .

A continuación, definimos, para cada clave  $k$ , la variable aleatoria  $Y_k$  que es igual al número de claves distintas de  $k$  que se encuentran en la misma ranura que  $k$ , de modo que

$$Y_k = \sum_{\substack{I \in T \\ I \neq k}} X_{Ik}$$

Así tenemos

$$E[Y_k] = E[\sum_{\substack{I \in T \\ I \neq k}} X_{Ik}] = \sum_{\substack{I \in T \\ I \neq k}} E[X_{Ik}]$$

$$\begin{aligned} &= \sum_{\substack{I \in T \\ I \neq k}} \frac{1}{m} \cdot m = \sum_{\substack{I \in T \\ I \neq k}} 1 = m - 1 \end{aligned}$$

(por linealidad de la expectativa)

El resto de la prueba depende de si la clave  $k$  está en la tabla  $T$ .

Si  $k \in T$ , entonces  $n_{h,k} / D = Y_k + j_f(l) - 1$  y  $l \neq k$ . Así  $E[n_{h,k} / D] = E[Y_k] + 1$ .

Si  $k \notin T$ , entonces como la clave  $k$  aparece en la lista  $T$   $E[n_{h,k} / D] = E[Y_k]$  no incluye la clave  $k$ , tenemos  $n_{h,k} / D = Y_k + j_f(l) - 1$  y  $l \neq k$ .

Así  $E[n_{h,k} / D] = E[Y_k] + 1$ .

El siguiente corolario dice que el hashing universal proporciona la recompensa deseada: ahora se ha vuelto imposible para un adversario elegir una secuencia de operaciones que fuerce el tiempo de ejecución del peor de los casos. Al aleatorizar hábilmente la elección de la función hash en tiempo de ejecución, garantizamos que podemos procesar cada secuencia de operaciones con un buen tiempo de ejecución de caso promedio.

#### Corolario 11.4 Al

usar el hash universal y la resolución de colisiones mediante el encadenamiento en una tabla inicialmente vacía con  $m$  ranuras, se necesita el tiempo esperado  $O(n/m)$  para manejar cualquier secuencia de  $n$  operaciones  $\text{INSERT}$ ,  $\text{SEARCH}$  y  $\text{DELETE}$  que contengan operaciones  $O(m/\sqrt{n})$ .

Prueba Dado que el número de inserciones es  $O(m/\sqrt{n})$ , tenemos  $n = O(m^2/\sqrt{n})$  y por lo tanto  $O(1/\sqrt{n})$ . Las operaciones  $\text{INSERTAR}$  y  $\text{ELIMINAR}$  toman un tiempo constante  $O(1)$ , según el teorema 11.3, el tiempo esperado para cada operación de  $\text{BÚSQUEDA}$  es  $O(1/\sqrt{n})$ . Por linealidad de

expectativa, por lo tanto, el tiempo esperado para toda la secuencia de  $n$  operaciones es  $O(n)$ . Dado que cada operación toma  $\frac{1}{n}$  tiempo, se sigue el límite  $\frac{1}{n}$ .

### Diseño de una clase universal de funciones hash

Es bastante fácil diseñar una clase universal de funciones hash, como nos ayudará a probar un poco de teoría numérica. Es posible que desee consultar primero el Capítulo 31 si no está familiarizado con la teoría de números.

Comenzamos eligiendo un número primo  $p$  lo suficientemente grande para que cada posible clave  $k$  esté en el rango de 0 a  $p-1$ , inclusive. Sea  $Z_p$  el conjunto  $\{0, 1, \dots, p-1\}$ , denote el  $y$  sea  $Z_p^m$  el conjunto  $\{0, 1, \dots, p-1\}^m$ . Dado que  $p$  es primo, podemos resolver ecuaciones módulo  $p$  con los métodos dados en el Capítulo 31. Dado que suponemos que el tamaño del universo de claves es mayor que el número de ranuras en la tabla hash, tenemos  $p > m$ .

Ahora definimos la función hash  $h_{ab}$  para cualquier  $a \in Z_p^m$  y cualquier  $b \in Z_p$  usando una transformación lineal seguida de reducciones módulo  $p$  y luego módulo  $m$ :

$$h_{ab}(a) = b a \mod p \quad (11.3)$$

Por ejemplo, con  $p=17$  y  $m=6$ , tenemos  $h_{3,8}(4,8) = 5$ . La familia de todas estas funciones hash es

$$H_{pm} = \{h_{ab} : a \in Z_p^m, b \in Z_p\} \quad (11.4)$$

Cada función hash  $h_{ab}$  asigna  $Z_p$  a  $Z_m$ . Esta clase de funciones hash tiene la buena propiedad de que el tamaño  $m$  del rango de salida es arbitrario, no necesariamente primo, una característica que usaremos en la Sección 11.5. Dado que tenemos  $p-1$  opciones para  $a$  y  $p$  opciones para  $b$ , la colección  $H_{pm}$  contiene funciones hash  $p^{p-1}$ .

### Teorema 11.5

La clase  $H_{pm}$  de funciones hash definidas por las ecuaciones (11.3) y (11.4) es universal.

Prueba Considere dos claves distintas  $k$  y  $l$  de  $Z_p$ , de modo que  $k \neq l$ . Para una función hash dada  $h_{ab}$  dejamos

$$r = h_{ab}(k) \mod p ; s = h_{ab}(l) \mod p$$

$$b \mod p :$$

Primero observamos que  $r \neq s$ . ¿Por qué? Observa eso

$$rs \equiv ak \mod p$$

De ello se deduce que  $r \neq s$  porque  $p$  es primo y tanto  $a$  como  $kl$  son módulos  $p$  distintos de cero, por lo que su producto también debe ser módulo  $p$  distinto de cero por el teorema 31.6. Por lo tanto, al calcular cualquier  $h_{ab}$  las distintas entradas  $k$  y  $l$  se asignan a distintas

valores r y s módulo p; todavía no hay colisiones en el "nivel mod p". Además, cada una de las opciones posibles de  $p.p1/$  para el par .a; b/ con  $a \neq 0$  produce un par resultante diferente .r; s/ con  $r \neq s$ , ya que podemos resolver para a y b dados r y s: a D .rs/..kl/1

$\text{mod } p / \text{mod } p ; b D .r ak / \text{mod } p$  ; donde ..kl/1 mod

$p/$  denota el inverso

multiplicativo único, módulo p, de k l. Como solo hay  $pp 1/$  pares posibles .r; s/ con  $r \neq s$ , existe una correspondencia biunívoca entre los pares .a; b/ con  $a \neq 0$  y pares .r; s/ con  $r \neq s$ . Así, para cualquier par dado de entradas k y l, si elegimos .a; b/ uniformemente al azar de  $Z Zp$ , el par resultante .r; s/ es igualmente probable que sea cualquier par de

pag  
valores distintos módulo p.

Por lo tanto, la probabilidad de que las distintas claves k y l colisionen es igual a la probabilidad de que  $rs .mod m/$  cuando r y s se eligen aleatoriamente como valores distintos módulo p. Para un valor dado de r, de los  $p 1$  posibles valores restantes para s, el número de valores s tales que  $s \neq r$  y  $sr .mod m/$  es como máximo

$$dp=yo 1 \dots p C m 1 / =m / 1 \text{ (por desigualdad (3.6))}$$

$$D .p 1 / =m :$$

La probabilidad de que s choque con r cuando se reduce el módulo m es como mucho  $.p 1 / =m / =.p 1 / D 1 =m$ .

Por lo tanto, para cualquier par de valores distintos k; l 2

$Zp, Pr fhab.k/ D hab.l/g 1=m$  ; por

lo que  $Hpm$  es de hecho universal. ■

## Ejercicios

### 11.3-1

Supongamos que deseamos buscar en una lista enlazada de longitud n, donde cada elemento contiene una clave k junto con un valor hash hk/. Cada clave es una larga cadena de caracteres. ¿Cómo podríamos aprovechar los valores hash al buscar en la lista un elemento con una clave dada?

### 11.3-2

Suponga que hacemos hash de una cadena de r caracteres en m espacios tratándolos como un número de raíz-128 y luego usando el método de división. Podemos representar fácilmente el número m como una palabra de computadora de 32 bits, pero la cadena de r caracteres, tratada como un número de base 128, requiere muchas palabras. ¿Cómo podemos aplicar el método de división para calcular el valor hash de la cadena de caracteres sin usar más que un número constante de palabras de almacenamiento fuera de la propia cadena?

## 11.3-3

Consideré una versión del método de división en la que  $hk \equiv k \bmod m$ , donde  $m \leq 2^p - 1$  y  $k$  es una cadena de caracteres interpretada en base  $2^p$ . Muestre que si podemos derivar la cadena  $x$  de la cadena  $y$  permutando sus caracteres, entonces  $x$  e  $y$  tienen el mismo valor. Dé un ejemplo de una aplicación en la que esta propiedad sería indeseable en una función hash.

## 11.3-4

Consideré una tabla hash de tamaño  $m \leq 1000$  y una función hash correspondiente  $hk \equiv b_m \dots b_1 \bmod 1/c$  para  $A \in \{0, 1\}^c$ . Calcule las ubicaciones a las que se asignan las claves 61, 62, 63, 64 y 65.

## 11.3-5 ?

Defina una familia  $H$  de funciones hash desde un conjunto finito  $U$  hasta un conjunto finito  $B$  para que sea  $\epsilon$ -universal si para todos los pares de elementos distintos  $k$  y  $l$  en  $U$ ,

$$\Pr_{h \in H} h(k) = h(l)$$

donde la probabilidad es sobre la elección de la función hash  $h$  extraída al azar de la familia  $H$ . Demuestre que una familia universal de funciones hash debe tener

$$\frac{1}{|B|} \leq \frac{1}{|U|}$$

## 11.3-6 ?

Sea  $U$  el conjunto de  $n$ -tuplas de valores extraídos de  $Z_p^n$ , y sea  $B \subseteq Z_p$ , donde  $p$  es primo. Defina la función hash  $hb: U \rightarrow B$  para  $b \in Z_p$  en una  $n$ -tupla de entrada  $(a_0; a_1; \dots; a_{n-1}) \in U$  como

$$hb(a_0; a_1; \dots; a_{n-1}) = \sum_{j=0}^{n-1} a_j b^j \pmod{p}$$

y sea  $HD(hb) = \{b \in Z_p : \exists a \in U \text{ s.t. } hb(a) = b\}$ . Argumente que  $H$  es  $\epsilon$ -universal de acuerdo con la definición de  $\epsilon$ -universal del ejercicio 11.3-5. (Sugerencia: vea el ejercicio 31.4-4.)

## 11.4 Direccionamiento abierto

En el direccionamiento abierto, todos los elementos ocupan la propia tabla hash. Es decir, cada entrada de la tabla contiene un elemento del conjunto dinámico o NIL. Cuando buscamos un elemento, examinamos sistemáticamente las ranuras de la tabla hasta que encontramos el elemento deseado o nos aseguramos de que el elemento no está en la tabla. Sin listas y

no se almacenan elementos fuera de la tabla, a diferencia del encadenamiento. Por lo tanto, en el direccionamiento abierto, la tabla hash puede "llenarse" de modo que no se puedan realizar más inserciones; una consecuencia es que el factor de carga  $\alpha$  nunca puede exceder de 1.

Por supuesto, podríamos almacenar las listas enlazadas para encadenarlas dentro de la tabla hash, en los espacios de la tabla hash que de otro modo no se utilizarían (vea el ejercicio 11.2-4), pero la ventaja del direccionamiento abierto es que evita los punteros por completo. En lugar de seguir punteros, calculamos la secuencia de espacios a examinar. La memoria adicional liberada al no almacenar punteros proporciona a la tabla hash una mayor cantidad de ranuras para la misma cantidad de memoria, lo que potencialmente produce menos colisiones y una recuperación más rápida.

Para realizar la inserción utilizando direccionamiento abierto, examinamos o probamos sucesivamente la tabla hash hasta que encontramos una ranura vacía en la que colocar la clave. En lugar de fijarse en el orden  $0; 1; \dots; m-1$  (que requiere  $\Theta(n)$  tiempo de búsqueda), la secuencia de posiciones sondeadas depende de la llave que se inserte. Para determinar qué ranuras sondear, ampliamos la función hash para incluir el número de sonda (a partir de 0) como una segunda entrada. Por lo tanto, la función hash se convierte en

```
h WU f0; 1; : : : ; m 1g! f0; 1; : : : ; m 1g :
```

Con direccionamiento abierto, requerimos que para cada tecla  $k$ , la secuencia de prueba

```
hhk; 0/; hk; 1/; : : : ; hk; m 1/y0
```

sea una permutación de  $h0; 1; \dots; m-1$ , de modo que cada posición de la tabla hash finalmente se considere como una ranura para una nueva clave a medida que la tabla se llena. En el siguiente pseudocódigo, asumimos que los elementos en la tabla hash  $T$  son claves sin información de satélite; la clave  $k$  es idéntica al elemento que contiene la clave  $k$ . Cada ranura contiene una clave o NIL (si la ranura está vacía). El procedimiento HASH-INSERT toma como entrada una tabla hash  $T$  y una clave  $k$ . Devuelve el número de ranura donde almacena la clave  $k$  o marca un error porque la tabla hash ya está llena.

```
HASH-INSERT.T; k / i D 0
```

1    2 repetir

j D hk; i / 3 si

    T  $\ominus$ e<sub>j</sub> == NIL T

    4 Ej D k devuelve j otra cosa i

    5           D i C 1 8

    6           hasta que

    7           i == m

9 error "desbordamiento de tabla hash"

El algoritmo para buscar la clave  $k$  sondea la misma secuencia de ranuras que el algoritmo de inserción examinó cuando se insertó la clave  $k$ . Por lo tanto, la búsqueda puede

termina (sin éxito) cuando encuentra una ranura vacía, ya que k se habría insertado allí y no más tarde en su secuencia de sondeo. (Este argumento supone que las claves no se eliminan de la tabla hash). El procedimiento HASH-SEARCH toma como entrada una tabla hash T y una clave k, devolviendo j si encuentra que la ranura j contiene la clave k, o NIL si la clave k es no presente en la tabla T

```
HASH-BÚSQUEDA.T; k/
1 i D 0 2
repetir 3 j D
hk; i / 4 si T [j] == k 5
regresa j 6 i D i C 1 7
hasta que T [j] = = NIL o i
= = m 8 regresa NIL
```

La eliminación de una tabla hash de direcciones abiertas es difícil. Cuando eliminamos una clave de la ranura i, no podemos simplemente marcar esa ranura como vacía almacenando NIL en ella. Si lo hiciéramos, es posible que no podamos recuperar ninguna clave k durante cuya inserción hayamos sondeado la ranura i y la hayamos encontrado ocupada. Podemos solucionar este problema marcando el slot, almacenando en él el valor especial DELETED en lugar de NIL. Entonces modificaríamos el procedimiento HASH-INSERT para tratar dicha ranura como si estuviera vacía para que podamos insertar una nueva clave allí. No necesitamos modificar HASH-SEARCH, ya que pasará por encima de los valores ELIMINADOS durante la búsqueda. Sin embargo, cuando usamos el valor especial DELETED, los tiempos de búsqueda ya no dependen del factor de carga  $\alpha$ , y, por esta razón, el encadenamiento se selecciona más comúnmente como una técnica de resolución de colisiones cuando se deben eliminar las claves.

En nuestro análisis, asumimos un hashing uniforme: la secuencia de prueba de cada clave tiene la misma probabilidad de ser cualquiera de las permutaciones  $m!/\alpha^m$  de  $h_0; 1; \dots; m$ . Una forma de hashing generaliza la noción de hash uniforme simple definida anteriormente a una función hash que produce no solo un número, sino una secuencia de prueba completa.

Sin embargo, es difícil implementar un hashing verdaderamente uniforme y, en la práctica, se utilizan aproximaciones adecuadas (como el hashing doble, que se define a continuación).

Examinaremos tres técnicas comúnmente utilizadas para calcular las secuencias de sondeo requeridas para el direccionamiento abierto: sondeo lineal, sondeo cuadrático y hash doble. Todas estas técnicas garantizan que  $h_k; h_{k+1}; \dots; h_{k+m-1}$  es una permutación de  $h_0; h_1; \dots; h_m$  para cada tecla  $k$ . Sin embargo, ninguna de estas técnicas cumple el supuesto de hash uniforme, ya que ninguna de ellas es capaz de generar más de  $m^2$  de secuencias de sondeo diferentes (en lugar de las  $m!$  que requiere el hashing uniforme). El hashing doble tiene el mayor número de secuencias de sondeo y, como era de esperar, parece dar los mejores resultados.

### sondeo lineal

Dada una función hash ordinaria  $h_0 \rightarrow \{0, 1, \dots, m-1\}$ , a la que nos referimos como función hash auxiliar, el método de sondeo lineal utiliza la función hash

$$h_k; i \rightarrow h_0(k) + i \bmod m$$

para  $i \in \{0, 1, \dots, m-1\}$ . Dada la clave  $k$ , primero probamos  $T \in h_0(k)$ , es decir, la ranura dada por la función hash auxiliar. A continuación, sondeamos la ranura  $T \in h_0(k) + 1 \bmod m$ , y así sucesivamente hasta la ranura  $T \in h_0(k) + m - 1 \bmod m$ . Luego envolvemos las ranuras  $T \in h_0(k) + i \bmod m$  hasta que finalmente probemos la ranura  $T \in h_0(k) + 1 \bmod m$ . Dado que el sondeo inicial determina la secuencia de sondeo completa, solo hay  $m$  secuencias de sondeo distintas.

El sondeo lineal es fácil de implementar, pero adolece de un problema conocido como agrupamiento primario. Se acumulan largas tiradas de espacios ocupados, lo que aumenta el tiempo de búsqueda promedio. Los grupos surgen porque una ranura vacía precedida por  $i$  ranuras llenas se llena a continuación con probabilidad  $i/m$ . Las tiradas largas de espacios ocupados tienden a alargarse y el tiempo medio de búsqueda aumenta.

### sondeo cuadrático

El sondeo cuadrático utiliza una función hash de la forma

$$h_k; i \rightarrow h_0(k) + c_1 i + c_2 i^2 \bmod m \quad (11.5)$$

donde  $h_0$  es una función hash auxiliar,  $c_1$  y  $c_2$  son constantes auxiliares positivas e  $i \in \{0, 1, \dots, m-1\}$ . La posición inicial palpada es  $T \in h_0(k)$ ; Las posiciones posteriores sondeadas se compensan con cantidades que dependen de forma cuadrática del número de sonda  $i$ . Este método funciona mucho mejor que el sondeo lineal, pero para aprovechar al máximo la tabla hash, los valores de  $c_1$ ,  $c_2$  y  $m$  están restringidos. El problema 11-3 muestra una forma de seleccionar estos parámetros. Además, si dos teclas tienen la misma posición de sondeo inicial, entonces sus secuencias de sondeo son las mismas, ya que  $h(k_1) / D \cdot h(k_2) / D \bmod m$  implica  $h(k_1)/D = h(k_2)/D$ . Esta propiedad conduce a una forma más leve de agrupamiento, denominada agrupamiento secundario. Como en el sondeo lineal, el sondeo inicial determina la secuencia completa, por lo que solo se utilizan  $m$  secuencias de sondeo distintas.

### Hash doble

El hashing doble ofrece uno de los mejores métodos disponibles para el direccionamiento abierto porque las permutaciones producidas tienen muchas de las características de las permutaciones elegidas al azar. El hashing doble utiliza una función hash de la forma

$$h_k; i \rightarrow h_1(k) + h_2(k) \bmod m$$

donde tanto  $h_1$  como  $h_2$  son funciones hash auxiliares. El palpador inicial va a la posición  $T \in h_1(k)$ ; Las posiciones sucesivas de la sonda se compensan con las posiciones anteriores por el

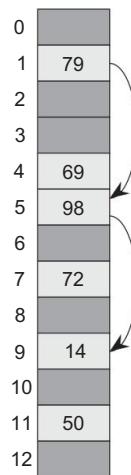


Figura 11.5 Inserción por doble hashing. Aquí tenemos una tabla hash de tamaño 13 con  $h_1(k) \equiv k \pmod{13}$  y  $h_2(k) \equiv 1 - C \cdot k \pmod{11}$ . Dado que  $14 \equiv 1 \pmod{13}$  y  $14 \equiv 3 \pmod{11}$ , insertamos la llave 14 en la ranura vacía 9, después de examinar las ranuras 1 y 5 y encontrarlas ocupadas.

cantidad  $h_2(k)$ , módulo  $m$ . Por lo tanto, a diferencia del caso del sondeo lineal o cuadrático, la secuencia de sondeo aquí depende de dos maneras de la clave  $k$ , ya que la posición inicial del sondeo, el desplazamiento o ambos pueden variar. La Figura 11.5 da un ejemplo de inserción por doble hashing.

El valor  $h_2(k)$  debe ser relativamente primo para el tamaño de la tabla hash  $m$  para que se busque en toda la tabla hash. (Vea el ejercicio 11.4-4.) Una forma conveniente de asegurar esta condición es hacer que  $m$  sea una potencia de 2 y diseñar  $h_2$  de modo que siempre produzca un número impar. Otra forma es dejar que  $m$  sea primo y diseñar  $h_2$  para que siempre devuelva un número entero positivo menor que  $m$ . Por ejemplo, podríamos elegir  $m$  prima y dejar

$h_1(k) \equiv k \pmod{m}$   $h_2(k) \equiv D$  :  
 $1 - C \cdot k \pmod{m} / ;$

donde  $m_0$  se elige ligeramente menor que  $m$  (por ejemplo,  $m = 1$ ). Por ejemplo, si  $k \equiv 123456 \pmod{m}$  y  $D \equiv 701 \pmod{m}$ , tenemos  $h_1(k) \equiv 80 \pmod{m}$  y  $h_2(k) \equiv 257 \pmod{m}$ , por lo que primero sondeamos la posición 80 y luego examinamos cada ranura 257 (módulo  $m$ ) hasta que encontramos la llave o hayamos examinado todas las ranuras.

Cuando  $m$  es primo o una potencia de 2, el hash doble mejora el sondeo lineal o cuadrático en el sentido de que se utilizan secuencias de sondeo  $\{m\}$ , en lugar de  $\{m^k\}$ , ya que cada posible  $h_1(k)$ ; El par  $h_2(k)$  produce una secuencia de sonda distinta. Como resultado, por

tales valores de m, el desempeño del hash doble parece estar muy cerca del desempeño del esquema "ideal" de hashing uniforme.

Aunque los valores de m que no sean primos o potencias de 2 se podrían usar en principio con doble hashing, en la práctica se vuelve más difícil generar eficientemente h2.k/ de una manera que asegure que es relativamente primo con m, en parte porque la densidad relativa .m/m de tales números puede ser pequeña (ver ecuación (31.24)).

#### Análisis de hashing de direcciones abiertas

Al igual que en nuestro análisis de encadenamiento, expresamos nuestro análisis de direccionamiento abierto en términos del factor de carga , D n=m de la tabla hash. Por supuesto, con el direccionamiento abierto, como máximo un elemento ocupa cada ranura y, por lo tanto, nm, lo que implica , 1.

Suponemos que estamos usando hashing uniforme. En este esquema idealizado, la secuencia de la sonda hh.k; 0/; hk; 1/; : : : ; hk; m 1/i utilizado para insertar o buscar cada clave k es igualmente probable que sea cualquier permutación de h0; 1; : : : ; m 1i. Por supuesto, una clave determinada tiene asociada una secuencia de sondeo fija única; lo que queremos decir aquí es que, considerando la distribución de probabilidad en el espacio de claves y la operación de la función hash en las claves, cada posible secuencia de prueba es igualmente probable.

Ahora analizamos el número esperado de sondajes para hash con direccionamiento abierto bajo el supuesto de hashing uniforme, comenzando con un análisis del número de sondajes realizados en una búsqueda fallida.

#### Teorema 11.6

Dada una tabla hash de direcciones abiertas con un factor de carga , D n=m < 1, el número esperado de sondajes en una búsqueda fallida es como máximo 1 = 0,1/, suponiendo un hash uniforme.

Prueba En una búsqueda fallida, todos los sondajes, excepto el último, acceden a un espacio ocupado que no contiene la clave deseada, y el último espacio sondeado está vacío. Definimos la variable aleatoria X como el número de sondajes realizados en una búsqueda fallida, y definimos también el evento Ai, para i 1; 2; : : : , para ser el evento de que se produce un i-ésimo sondeo y se trata de una ranura ocupada. Entonces el evento fX ig es la intersección de los eventos A1\A2\ \Ai1. Acotaremos Pr fX ig acotando Pr fA1 \ A2 \ \ Ai1g. Por el ejercicio C.2-5,

$$\Pr{fA1 \setminus A2 \setminus \cup_{i=1}^j A_i} \leq \Pr{fA1} \Pr{fA2 \setminus \cup_{i=1}^{j-1} A_i} \Pr{fA3 \setminus \cup_{i=1}^{j-2} A_i} \dots \Pr{fA_{j-1} \setminus \cup_{i=1}^{j-2} A_i}$$

$$\Pr{fA1} = \frac{1}{m}, \quad \Pr{fA2} = \frac{1}{m}, \quad \Pr{fA3} = \frac{1}{m}, \quad \dots, \quad \Pr{fA_{j-1}} = \frac{1}{m}$$

Como hay n elementos y m ranuras,  $\Pr{fA1} = \frac{1}{m}$ . Para  $j > 1$ , la probabilidad de que haya una j-ésima sonda y sea en un slot ocupado, dado que las primeras  $j-1$  sondas fueron en slots ocupados, es  $\frac{j}{n} \cdot \frac{n-j}{m-1}$ . Esta probabilidad sigue

porque estaríamos encontrando uno de los elementos  $.n .j 1//$  restantes en una de las ranuras  $.m .j 1//$  no examinadas, y por la suposición de hash uniforme, la probabilidad es la proporción de estas cantidades. Observando que  $n < m$  implica que  $.nj / = .mj / n = m$  para todo  $j$  tal que  $0 \leq j < m$ , tenemos para todo  $i$  tal que  $1 \leq i \leq n$ ,

$$\Pr_{fX \text{ igD}} \left[ \begin{array}{c} \text{norte} \\ \text{metro} \\ \hline \text{norte} \\ \text{metro} \end{array} \middle| \begin{array}{c} n \ 1 \\ \text{metro } 1 \\ \hline n^o \ 2 \\ \text{metro } 2 \\ \hline n_i \ C \ 2 \\ \text{mi } C \ 2 \end{array} \right] = \frac{1}{n}$$

$D_{j1}$

Ahora, usamos la ecuación (C.25) para acotar el número esperado de sondas:

$$E \ OEX \ D \ X1 = \Pr_{fX \text{ igD}}_{iD1}$$

$$X1^{i1}_{iD1}$$

$$X1 = \frac{i}{iD0} = \frac{1}{1}$$

■

Este límite de  $1 = .1 / D$   $1C, C, 2C, 3C$  tiene una interpretación intuitiva. Siempre hacemos la primera sonda. Con una probabilidad de aproximadamente  $\alpha$ , el primer sondeo encuentra un espacio ocupado, por lo que necesitamos sondear una segunda vez. Con una probabilidad aproximada de  $\beta$ , las dos primeras ranuras están ocupadas, de modo que hacemos una tercera sonda, y así sucesivamente.

Si  $\alpha$  es una constante, el teorema 11.6 predice que una búsqueda fallida se ejecuta en un tiempo  $O(1)$ . Por ejemplo, si la tabla hash está llena hasta la mitad, la cantidad promedio de sondeos en una búsqueda fallida es como máximo  $1 = .5 / D$ . Si está llena en un 90 por ciento, la cantidad promedio de sondeos es como máximo  $1 = .1 / D$ .

El teorema 11.6 nos da el desempeño del procedimiento HASH-INSERT casi inmediatamente.

#### Corolario 11.7

Insertar un elemento en una tabla hash de dirección abierta con factor de carga  $\alpha$  requiere como máximo  $1 = 0,1 / \alpha$  sondeos en promedio, suponiendo un hash uniforme.

Prueba Un elemento se inserta solo si hay espacio en la tabla, y por lo tanto  $\leq 1$ . Insertar una clave requiere una búsqueda fallida seguida de colocar la clave en la primera ranura vacía encontrada. Por tanto, el número esperado de sondajes es como máximo  $1 = 0, \frac{1}{1}$ .

Tenemos que hacer un poco más de trabajo para calcular el número esperado de sondajes para una búsqueda exitosa.

### Teorema 11.8

Dada una tabla hash de direcciones abiertas con factor de carga  $\leq 1$ , el número esperado de sondajes en una búsqueda exitosa es como máximo

$$\text{4 en } \frac{1}{1} :$$

asumiendo un hashing uniforme y asumiendo que cada clave en la tabla tiene la misma probabilidad de ser buscada.

Prueba Una búsqueda de una clave  $k$  reproduce la misma secuencia de sondeo que cuando se insertó el elemento con la clave  $k$ . Por el Corolario 11.7, si  $k$  era la clave  $i$  C 1/st insertada en la tabla hash, el número esperado de sondajes realizados en una búsqueda de  $k$  es como máximo  $1 = .1 i=m/ D m=.mi/$ . Promediar todas las  $n$  claves en la tabla hash nos da el número esperado de sondajes en una búsqueda exitosa:

$$\begin{aligned}
 & \frac{1}{n} X_{n1} \overbrace{\text{metro}}^{\text{northeast}} \quad \text{D} \quad \frac{1}{n} X_{n1} \overbrace{\text{metro}}^{\text{northeast}} \frac{1}{m} \\
 & \text{D} \quad \frac{1}{k} X_m \frac{1}{k} \\
 & \frac{1}{Z} \overbrace{\text{Minneapolis}}^{\text{metropolitan}} .1=x/ dx \text{ (por desigualdad (A.12))} \\
 & \text{D} \quad \frac{1}{\min 1} \\
 & \text{D} \quad \frac{1}{1} :
 \end{aligned}$$

Si la tabla hash está llena hasta la mitad, el número esperado de sondajes en una búsqueda satisfactoria es inferior a 1:387. Si la tabla hash está llena en un 90 por ciento, el número esperado de sondajes es inferior a 2:559.

**Ejercicios****11.4-1**

Considere insertar las llaves 10; 22; 31; 4; 15; 28; 17; 88; 59 en una tabla hash de longitud  $m = D = 11$  usando direccionamiento abierto con la función hash auxiliar  $h_0(k) = k \mod D$ .

Ilustre el resultado de insertar estas claves usando sondeo lineal, usando sondeo cuadrático con  $c_1 = D - 1$  y  $c_2 = D - 3$ , y usando hash doble con  $h_1(k) = D - k$  y  $h_2(k) = D - 1 - k \mod m = 11 - k$ .

**11.4-2**

Escriba el pseudocódigo para HASH-DELETE como se describe en el texto y modifique HASH-INSERT para manejar el valor especial DELETED.

**11.4-3**

Considere una tabla hash de direcciones abiertas con hash uniforme. Proporcione los límites superiores del número esperado de sondeos en una búsqueda fallida y del número esperado de sondeos en una búsqueda exitosa cuando el factor de carga es  $3=4$  y cuando es  $7=8$ .

**11.4-4 ?**

Supongamos que usamos hash doble para resolver colisiones, es decir, usamos la función hash  $h(k) = i \cdot D \cdot h_1(k) + h_2(k) \mod m$ . Muestre que si  $m$  y  $h_2(k)$  tienen el máximo común divisor  $d > 1$  para alguna clave  $k$ , entonces una búsqueda fallida de la clave  $k$  examina  $i = d \cdot t/h$  de la tabla hash antes de regresar a la ranura  $h_1(k)$ . Por lo tanto, cuando  $d = 1$ , de modo que  $m$  y  $h_2(k)$  son primos relativos, la búsqueda puede examinar toda la tabla hash. (Sugerencia: consulte el Capítulo 31.)

**11.4-5 ?**

Considere una tabla hash de direcciones abiertas con un factor de carga  $\alpha$ . Encuentre el valor distinto de cero  $\alpha$  para el cual el número esperado de sondeos en una búsqueda fallida es igual al doble del número esperado de sondeos en una búsqueda exitosa. Use los límites superiores dados por los teoremas 11.6 y 11.8 para estos números esperados de sondas.

**? 11.5 Hash perfecto**

Aunque el hashing suele ser una buena opción por su excelente rendimiento en el caso promedio, el hashing también puede proporcionar un excelente rendimiento en el peor de los casos cuando el conjunto de claves es estático: una vez que las claves se almacenan en la tabla, el conjunto de claves nunca cambia. Algunas aplicaciones, naturalmente, tienen conjuntos estáticos de claves: considere el conjunto de palabras reservadas en un lenguaje de programación, o el conjunto de nombres de archivo en un CD-ROM. Nosotros

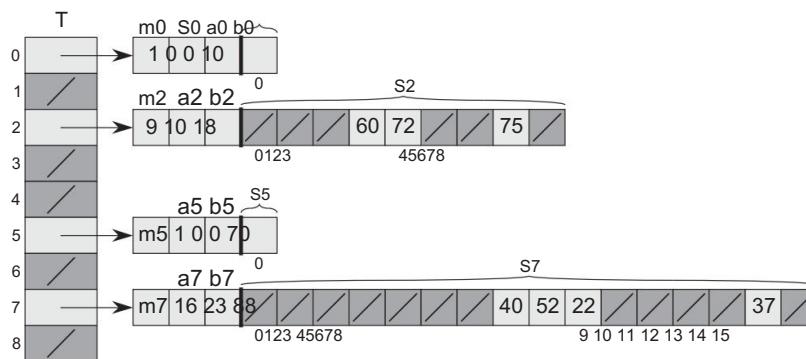


Figura 11.6 Uso de hashing perfecto para almacenar el conjunto KD f10; 22; 37; 40; 52; 60; 70; 72; 75g. La función hash externa es  $hk / D ..ak C b / mod p / mod m$ , donde a D 3, b D 42, p D 101 y m D 9. Por ejemplo, h.75/ D 2, etc. key 75 hash en la ranura 2 de la tabla TA, la tabla hash secundaria Sj almacena todas las claves hash en la ranura j. El tamaño de la tabla hash Sj es  $mj D n2$  y la función hash asociada es  $hj .k / D ..aj k C bj / mod p / mod mj$ . Dado que  $h2.75 / D 7$ , la clave 75 se almacena en la ranura 7 de la tabla hash secundaria S2. No se producen colisiones en ninguna de las tablas hash secundarias, por lo que la búsqueda requiere un tiempo constante en el peor de los casos.

llame a una técnica hash hash perfecto si O.1/ se requieren accesos a la memoria para realizar una búsqueda en el peor de los casos.

Para crear un esquema de hash perfecto, utilizamos dos niveles de hashing, con hash universal en cada nivel. La figura 11.6 ilustra el enfoque.

El primer nivel es esencialmente el mismo que para el hash con encadenamiento: hash las n claves en m ranuras usando una función hash h cuidadosamente seleccionada de una familia de funciones hash universales.

Sin embargo, en lugar de hacer una lista enlazada de las claves que codifican para la ranura j, usamos una pequeña tabla hash secundaria Sj con una función hash asociada  $hj$ . Al elegir cuidadosamente las funciones hash  $hj$ , podemos garantizar que no haya colisiones en el nivel secundario.

Sin embargo, para garantizar que no haya colisiones en el nivel secundario, necesitaremos que el tamaño  $mj$  de la tabla hash Sj sea el cuadrado del número  $nj$  de claves hash en la ranura j. Aunque podría pensar que la dependencia cuadrática de  $mj$  con respecto a  $nj$  puede causar que el requisito de almacenamiento general sea excesivo, mostraremos que al elegir bien la función hash de primer nivel, podemos limitar la cantidad total esperada de espacio utilizado a encendido/.

Usamos funciones hash elegidas de las clases universales de funciones hash de la Sección 11.3.3. La función hash de primer nivel proviene de la clase Hpm, donde, como en la Sección 11.3.3, p es un número primo mayor que cualquier valor clave. esas llaves

hash a la ranura  $j$  se vuelven a convertir en una tabla hash secundaria  $S_j$  de tamaño  $m_j$  utilizando una función hash  $h_j$  elegida de la clase  $H_p; m_j$ .<sup>1</sup>

Procederemos en dos pasos. Primero, determinaremos cómo garantizar que las tablas secundarias no tengan colisiones. En segundo lugar, mostraremos que la cantidad esperada de memoria utilizada en general, para la tabla hash primaria y todas las tablas hash secundarias, es  $O(n)$ .

### Teorema 11.9

Supongamos que almacenamos  $n$  claves en una tabla hash de tamaño  $m \leq n^2$  utilizando una función hash  $h$  elegida aleatoriamente de una clase universal de funciones hash. Entonces, la probabilidad es menor que  $1=2$  de que haya colisiones.

Prueba Hay pares de llaves que pueden chocar; cada par choca con probabilidad  $1=m$  si  $h$  se elige al azar de una familia universal  $H$  de funciones hash.

Sea  $X$  una variable aleatoria que cuenta el número de colisiones. Cuando  $m \leq n^2$ , el número esperado de colisiones es

$$\begin{aligned} E[X] &= D \cdot \frac{n^2}{n^2} \\ &= D \cdot \frac{n^2}{n^2} = D \\ &= D \cdot \frac{1}{n^2} = \frac{D}{n^2} \end{aligned}$$

$2 < 1=2 :$

(Este análisis es similar al análisis de la paradoja del cumpleaños en la Sección 5.4.1.)

Aplicando la desigualdad de Markov (C.30),  $\Pr[X \geq t] \leq \frac{E[X]}{t}$ , con  $t = 1$ , completa la prueba. ■

En la situación descrita en el Teorema 11.9, donde  $m \leq n^2$ , se deduce que una función hash  $h$  elegida al azar de  $H$  tiene más probabilidades de no tener colisiones.

Dado el conjunto  $K$  de  $n$  claves que se van a codificar (recuerde que  $K$  es estático), es fácil encontrar una función hash libre de colisiones  $h$  con algunos intentos aleatorios.

Sin embargo, cuando  $n$  es grande, una tabla hash de tamaño  $m \leq n^2$  es excesiva. Por lo tanto, adoptamos el enfoque hash de dos niveles y usamos el enfoque del teorema 11.9 solo para hash las entradas dentro de cada ranura. Usamos una función hash externa, o de primer nivel,  $h$  para codificar las claves en ranuras  $m \leq n$ . Luego, si las claves  $n_j$  generan un hash en la ranura  $j$ , use una tabla hash secundaria  $S_j$  de tamaño  $m_j \leq D$  para proporcionar una búsqueda de tiempo constante sin colisiones.

---

<sup>1</sup> Cuando  $n_j \leq m_j \leq 1$ , en realidad no necesitamos una función hash para la ranura  $j$ ; cuando elegimos una función hash  $h_{ab,k} : D \rightarrow C$  para la ranura  $j$ , simplemente usamos  $b \in D$ .

Ahora pasamos a la cuestión de garantizar que la memoria general utilizada esté activada/. Dado que el tamaño  $m_j$  de la  $j$ -ésima tabla hash secundaria crece cuadráticamente con el número  $n_j$  de claves almacenadas, corremos el riesgo de que la cantidad total de almacenamiento sea excesiva.

Si el tamaño de la tabla de primer nivel es  $m \leq n$ , entonces la cantidad de memoria utilizada es  $O(n^2)$  para la tabla hash primaria, para el almacenamiento de los tamaños  $m_j$  de las tablas hash secundarias y para el almacenamiento de los parámetros  $a_j$  y  $b_j$  definiendo las funciones hash secundarias  $h_j$  extraídas de la clase  $H_{m_j}$  de la Sección 11.3.3 (excepto cuando  $n_j = 1$  y usamos  $a_j = b_j = 0$ ). El siguiente teorema y un corolario proporcionan un límite en los tamaños combinados esperados de todas las tablas hash secundarias. Un segundo corolario limita la probabilidad de que el tamaño combinado de todas las tablas hash secundarias sea superlineal (en realidad, que sea igual o superior a  $4n$ ).

#### Teorema 11.10

Suponga que almacenamos  $n$  claves en una tabla hash de tamaño  $m \leq n$  utilizando una función hash  $h$  elegida aleatoriamente de una clase universal de funciones hash. Entonces nosotros tenemos

$$\sum_{j=1}^{m-1} j^2 \leq 2n$$

donde  $n_j$  es el número de claves hash en la ranura  $j$ .

Prueba Comenzamos con la siguiente identidad, que se cumple para cualquier entero no negativo  $a$ :

$$\sum_{k=0}^{m-1} k^a = \frac{m^{a+1} - 1}{m - 1} \quad (11.6)$$

Tenemos

$$\sum_{j=1}^{m-1} j^2$$

$$= \sum_{j=0}^{m-1} j^2 = \frac{m^3 - m}{m - 1} \quad (\text{por la ecuación (11.6)})$$

$$= m \sum_{j=0}^{m-1} j^2 = m \cdot \frac{m^3 - m}{m - 1} \quad (\text{por linealidad de expectativa})$$

$$= m \cdot \frac{m^2(m-1)}{m-1} = m^3 \quad (\text{por la ecuación (11.1)})$$

$$D \ n \ C \ 2 \ E \ "m \ X_1 \ j \ b_0 \ 2 \ \# \quad (\text{ya que } n \text{ no es una variable aleatoria}).$$

Evaluar la suma  $Pm1$  de pares de claves en la tabla hash que chocan. Por las propiedades del hashing universal, el valor esperado de esta suma es como máximo

$$\begin{array}{c} \text{norte} \\ \text{---} \\ \text{2! 1} \\ \text{---} \\ \text{metro} \\ \text{---} \\ \text{n 1} \\ \text{---} \\ \text{2} \end{array}$$

desde  $m \geq n$ . De este modo,

$$\begin{array}{c} n^2 \\ \text{---} \\ E \ "m \ X_1 \ j \ # \\ \text{---} \\ D \ 2n \ 1 \\ \text{---} \\ < 2n \end{array}$$

■

### Corolario 11.11

Suponga que almacenamos  $n$  claves en una tabla hash de tamaño  $m \geq n$  usando una función hash  $h$  elegida al azar de una clase universal de funciones hash, y establecemos el tamaño de cada tabla hash secundaria en  $m_j \leq n^2$  para  $j = 0, 1, \dots, m-1$ . Entonces, la cantidad esperada de almacenamiento requerido para todas las tablas hash secundarias en un esquema hash perfecto es menor que  $2n$ .

Prueba Dado que  $m_j \leq n^2$  para  $j = 0, 1, \dots, m-1$ , el teorema 11.10 da

$$\begin{array}{c} n^2 \\ \text{---} \\ E \ "m \ X_1 \ m_j \ # \ DE \ "m \ X_1 \ j \ b_0 \ j \ # \\ \text{---} \\ < 2n \end{array} \quad (11.7)$$

■

que completa la demostración.

### Corolario 11.12

Suponga que almacenamos  $n$  claves en una tabla hash de tamaño  $m \geq n$  usando una función hash  $h$  elegida al azar de una clase universal de funciones hash, y establecemos el tamaño de cada tabla hash secundaria en  $m_j \leq n^2$  para  $j = 0, 1, \dots, m-1$ . Entonces, la probabilidad es menor que  $\frac{1}{2}$  de que el almacenamiento total utilizado para las tablas hash secundarias sea igual o exceda  $4n$ .

Demostración Nuevamente aplicamos la desigualdad de Markov (C.30),  $\Pr fX \geq E$   
 $\Pr X = t$ , esta vez a la desigualdad (11.7), con  $X \leq D$   $P_m \leq \frac{1}{D}$

$$\Pr(X \leq t) \leq \frac{\Pr(X \leq t)}{\Pr(X \leq 4n)} = \frac{2n}{4n} = \frac{1}{2}$$

D1=2: ■

Del Corolario 11.12, vemos que si probamos algunas funciones hash elegidas al azar de la familia universal, encontraremos rápidamente una que use una cantidad razonable de almacenamiento.

### Ejercicios

#### 11.5-1 ?

Supongamos que insertamos  $n$  claves en una tabla hash de tamaño  $m$  usando direccionamiento abierto y hash uniforme. Sea  $p_m/n$  sea la probabilidad de que no ocurran colisiones. Muestre que  $p_m/n \approx 2m$ . (Sugerencia: vea la ecuación (3.12).) Argumente que cuando  $n$  excede a  $pm$ , la probabilidad de evitar colisiones se reduce rápidamente a cero.

### Problemas

#### 11-1 Límite de sonda más largo para hashing

Supongamos que usamos una tabla hash de direcciones abiertas de tamaño  $m$  para almacenar  $nm$  elementos.

- Suponiendo hash uniforme, demuestre que para  $i \in \{1, 2, \dots, n\}$ , la probabilidad es como mucho  $2k$  de que la  $i$ -ésima inserción requiera estrictamente más de  $k$  sondas.
- Demuestre que para  $i \in \{1, 2, \dots, n\}$ , la probabilidad es  $O(1/n^2)$  de que la  $i$ -ésima inserción requiera más de  $2$  sondas  $\lg n$ .

Deje que la variable aleatoria  $X_i$  denote el número de sondas requeridas por la  $i$ -ésima inserción. Ha demostrado en la parte (b) que  $\Pr(X_i > 2 \lg n) \leq O(1/n^2)$ . Deje que la variable aleatoria  $D$   $\max_{1 \leq i \leq n} X_i$  denote el número máximo de sondas requeridas por cualquiera de las  $n$  inserciones.

C. Demuestre que  $\Pr(D > 2 \lg n) \leq O(1/n)$ .

d. Demuestre que la longitud esperada  $E(D)$  de la secuencia de prueba más larga es  $O(\lg n)$ .

**11-2 Límite de tamaño de ranura para encadenamiento** Suponga que tenemos una tabla hash con  $n$  ranuras, con colisiones resueltas por encadenamiento, y suponga que se insertan  $n$  claves en la tabla. Cada clave tiene la misma probabilidad de ser codificada en cada ranura. Sea  $M$  el número máximo de llaves en cualquier ranura después de haber insertado todas las llaves. Su misión es probar un límite superior  $O(\lg n = \lg \lg n)$  en  $E \text{C}\text{EM}$ , el valor esperado de  $M$ . a.

Argumente que la probabilidad  $Q_k$  de que exactamente  $k$  claves hagan hash en un espacio en particular es dada por

$$q_k = \frac{1}{n} \cdot \frac{k}{n} \cdot \frac{1}{n} \cdots \frac{1}{n} \cdot \frac{n-k}{n} = \frac{n!}{k!(n-k)!}$$

b. Sea  $P_k$  la probabilidad de que  $M \geq k$ , es decir, la probabilidad de que la ranura que contiene más claves contenga  $k$  claves. Demuestre que  $P_k \leq Q_k$ .

C. Use la aproximación de Stirling, ecuación (3.18), para mostrar que  $Q_k < e^{-ek} = kk^{-k}$ .

d. Demuestre que existe una constante  $c > 1$  tal que  $Q_{k0} < 1/n^3$  para  $k_0 \leq c \lg n = \lg \lg n$ . Concluya que  $P_k < 1/n^2$  para  $k \geq k_0$ .

mi. Argumenta eso

$$E \text{C}\text{EM} = \sum_{k=0}^{\infty} k P_k = \sum_{k=0}^{\infty} k \frac{c \lg n}{n} \cdot \frac{c \lg n}{n} \cdots \frac{c \lg n}{n} = \frac{c \lg n}{\lg \lg n} \cdot \frac{c \lg n}{\lg \lg n} \cdots \frac{c \lg n}{\lg \lg n}$$

Concluya que  $E \text{C}\text{EM} \leq O(\lg n = \lg \lg n)$ .

### 11-3 Sondeo cuadrático

Supongamos que se nos da una clave  $k$  para buscar en una tabla hash con posiciones  $0; 1; \dots; m-1$ , y supongamos que tenemos una función hash  $h$  mapeando el espacio clave en el conjunto  $f_0; f_1; \dots; f_{m-1}$ . El esquema de búsqueda es el siguiente:

- Calcular el valor  $j = h(k)$ , y establecer  $i = 0$ .

2. Sonda en posición  $j$  para la tecla deseada  $k$ . Si lo encuentra, o si esta posición es vacío, terminar la búsqueda.

3. Establezca  $i = i + 1$ . Si  $i$  ahora es igual a  $m$ , la tabla está llena, así que finalice la búsqueda. De lo contrario, configure  $j = (j + 1) \mod m$  y regrese al paso 2.

Suponga que  $m$  es una potencia de

2. a. Muestre que este esquema es un ejemplo del esquema general de "sondeo cuadrático" mostrando las constantes apropiadas  $c_1$  y  $c_2$  para la ecuación (11.5).

b. Demuestre que este algoritmo examina cada posición de la mesa en el peor de los casos.

11-4 Hashing y autenticación Sea  $H$  una clase de funciones hash en la que cada función hash  $h \in H$  asigna el universo  $U$  de claves a  $f_0; 1; \dots; p-1$ . Decimos que  $H$  es  $k$ -universal si, para toda secuencia fija de  $k$  claves distintas  $x_1, x_2, \dots, x_k$  y para cualquier  $h$  elegido al azar de  $H$ , la secuencia  $h(x_1); h(x_2); \dots; h(x_k)$  es igualmente probable que sea cualquiera de las secuencias  $m_k$  de longitud  $k$  con elementos extraídos de  $f_0; 1; \dots; p-1$ .

- Muestre que si la familia  $H$  de funciones hash es 2-universal, entonces es universal.
- Supongamos que el universo  $U$  es el conjunto de  $n$ -tuplas de valores extraídos de  $\mathbb{Z}_p$ :  $f_0; 1; \dots; p-1$ , donde  $p$  es primo. Considere un elemento  $x \in U$ :  $x = (x_0; x_1; \dots; x_{n-1}) \in U$ . Para cualquier  $n$ -tupla  $a \in U$ :  $a = (a_0; a_1; \dots; a_{n-1}) \in U$ , defina la función hash  $h_a$  por

$$h_a(x) = \sum_{j=0}^{n-1} a_j x_j \pmod{p}$$

Deja que  $H$  sea la familia de funciones hash. Muestre que  $H$  es universal, pero no 2-universal. (Sugerencia: encuentre una clave para la cual todas las funciones hash en  $H$  produzcan el mismo valor).

- Suponga que modificamos  $H$  ligeramente de la parte (b): para cualquier  $a \in U$  y para cualquier  $b \in \mathbb{Z}_p$ , definir

$$h_a(x) = \sum_{j=0}^{n-1} a_j x_j + b \pmod{p}$$

y  $H_0$  sea la familia de funciones hash. Argumente que  $H_0$  es 2-universal. (Sugerencia: Considere  $n$ -tuplas fijas  $x \in U$  y  $y \in U$ , con  $x_i \neq y_i$  para alguna  $i$ . ¿Qué sucede con  $h_0(x) \neq h_0(y)$  cuando  $a_i \neq b_i$  para algún  $i$ ?).

- Supongamos que Alice y Bob acuerdan en secreto una función hash  $h$  de una familia  $H$  universal de 2 funciones hash. Cada  $h \in H$  corresponde a un universo de claves  $U$  a  $\mathbb{Z}_p$ , donde  $p$  es primo. Más tarde, Alice envía un mensaje  $m$  a Bob a través de Internet, donde  $m \in U$ . Ella autentica este mensaje a Bob enviando también una etiqueta de autenticación  $t \in \mathbb{Z}_p$ , y Bob verifica que el par  $(m, t)$  satisface la ecuación  $t = h(m)$ . Supongamos que un adversario intercepta el par  $(m, t)$  en el camino y trata de engañar a Bob reemplazando el par  $(m, t)$  con un par diferente  $(m', t')$ .

Argumente que la probabilidad de que el adversario logre engañar a Bob para que acepte el par  $(m', t')$  es muy pequeña. Sin importar cuánto poder de cómputo tenga el adversario, e incluso si el adversario conoce la familia  $H$  de funciones hash utilizadas.

### Notas del capítulo

Knuth [211] y Gonnet [145] son excelentes referencias para el análisis de algoritmos hash. Knuth le da crédito a HP Luhn (1953) por inventar tablas hash, junto con el método de encadenamiento para resolver colisiones. Casi al mismo tiempo, GM Amdahl originó la idea del direccionamiento abierto.

Carter y Wegman introdujeron la noción de clases universales de funciones hash en 1979 [58].

Fredman, Koml'os y Szemer'edi [112] desarrollaron el esquema de hashing perfecto para conjuntos estáticos presentado en la Sección 11.5. Una extensión de su método a conjuntos dinámicos, manejando inserciones y eliminaciones en el tiempo esperado amortizado O.1/, ha sido dada por Dietzfelbinger et al. [86].

La estructura de datos del árbol de búsqueda admite muchas operaciones de conjuntos dinámicos, incluidas BUSCAR, MÍNIMO, MÁXIMO, PREDECESOR, SUCESOR, INSERTAR y ELIMINAR. Así, podemos usar un árbol de búsqueda tanto como diccionario como como prioridad cola.

Las operaciones básicas en un árbol de búsqueda binaria toman un tiempo proporcional a la altura del árbol. Para un árbol binario completo con  $n$  nodos, dichas operaciones se ejecutan en  $\lg n$  en el peor de los casos. Sin embargo, si el árbol es una cadena lineal de  $n$  nodos, las mismas operaciones toman  $n$  en el peor de los casos. Veremos en la Sección 12.4 que la altura esperada de un árbol de búsqueda binario construido aleatoriamente es  $O(\lg n)$ , de modo que las operaciones básicas de conjuntos dinámicos en dicho árbol toman  $\lg n$  de tiempo en promedio.

En la práctica, no siempre podemos garantizar que los árboles de búsqueda binarios se construyan aleatoriamente, pero podemos diseñar variaciones de árboles de búsqueda binarios con un buen rendimiento garantizado en el peor de los casos en operaciones básicas. El capítulo 13 presenta una de esas variaciones, árboles rojo-negros, que tienen una altura de  $O(\lg n)$ . El Capítulo 18 presenta los árboles B, que son particularmente buenos para mantener bases de datos en almacenamiento secundario (en disco).

Después de presentar las propiedades básicas de los árboles de búsqueda binaria, las siguientes secciones muestran cómo recorrer un árbol de búsqueda binaria para imprimir sus valores en orden, cómo buscar un valor en un árbol de búsqueda binaria, cómo encontrar el elemento mínimo o máximo, cómo encontrar el predecesor o el sucesor de un elemento y cómo insertarlo o eliminarlo de un árbol de búsqueda binaria. Las propiedades matemáticas básicas de los árboles aparecen en el Apéndice B.

---

## 12.1 ¿Qué es un árbol de búsqueda binario?

Un árbol de búsqueda binaria se organiza, como sugiere su nombre, en un árbol binario, como se muestra en la figura 12.1. Podemos representar dicho árbol mediante una estructura de datos enlazados en la que cada nodo es un objeto. Además de una clave y datos satelitales, cada nodo contiene atributos izquierdo, derecho y  $p$  que apuntan a los nodos correspondientes a su hijo izquierdo,

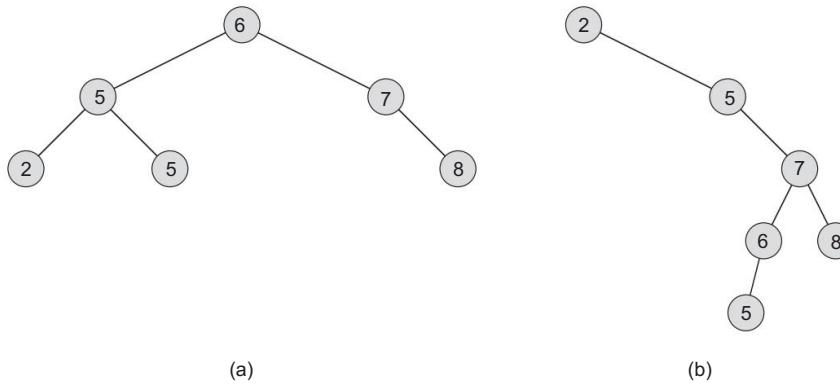


Figura 12.1 Árboles de búsqueda binarios. Para cualquier nodo  $x$ , las claves en el subárbol izquierdo de  $x$  son como máximo  $x:\text{clave}$ , y las claves en el subárbol derecho de  $x$  son al menos  $x:\text{clave}$ . Diferentes árboles de búsqueda binarios pueden representar el mismo conjunto de valores. El tiempo de ejecución en el peor de los casos para la mayoría de las operaciones del árbol de búsqueda es proporcional a la altura del árbol. (a) Un árbol de búsqueda binaria en 6 nodos con altura 2. (b) Un árbol de búsqueda binaria menos eficiente con altura 4 que contiene las mismas claves.

su hijo derecho y su padre, respectivamente. Si falta un elemento secundario o principal, el atributo apropiado contiene el valor NIL. El nodo raíz es el único nodo del árbol cuyo padre es NIL.

Las claves en un árbol de búsqueda binaria siempre se almacenan de tal manera que satisfagan la propiedad del árbol de búsqueda binaria:

Sea  $x$  un nodo en un árbol de búsqueda binaria. Si  $y$  es un nodo en el subárbol izquierdo de  $x$ , entonces  $y:\text{key} \leq x:\text{key}$ . Si  $y$  es un nodo en el subárbol derecho de  $x$ , entonces  $y:\text{key} \geq x:\text{key}$ .

Así, en la figura 12.1(a), la clave de la raíz es 6, las claves 2, 5 y 5 en su subárbol izquierdo no son mayores que 6, y las claves 7 y 8 en su subárbol derecho no son menores que 6. La misma propiedad es válida para todos los nodos del árbol. Por ejemplo, la clave 5 en el hijo izquierdo de la raíz no es más pequeña que la clave 2 en el subárbol izquierdo de ese nodo y no más grande que la clave 5 en el subárbol derecho.

La propiedad del árbol de búsqueda binaria nos permite imprimir todas las claves en un árbol de búsqueda binaria en orden mediante un algoritmo recursivo simple, llamado recorrido de árbol en orden. Este algoritmo se llama así porque imprime la clave de la raíz de un subárbol entre la impresión de los valores en su subárbol izquierdo y la impresión de los valores en su subárbol derecho.

(Del mismo modo, un recorrido de árbol de orden previo imprime la raíz antes de los valores en cualquiera de los subárboles, y un recorrido de árbol de orden posterior imprime la raíz después de los valores de sus subárboles). El siguiente procedimiento para imprimir todos los elementos en un árbol de búsqueda binario  $T$  es:

```

INORDER-TREE-WALK(T:root)
  if T ≠ NIL
    INORDER-TREE-WALK(T.left)
    print T.key
    INORDER-TREE-WALK(T.right)
  end
  
```

```

INORDER-TREE-WALK.x/ 1 if x
  ↘ NIL 2 INORDER-
    TREE-WALK.x:left/ 3 print x:key 4 INORDER-
      TREE-WALK.x:right/

```

Como ejemplo, el recorrido del árbol en orden imprime las claves en cada uno de los dos árboles de búsqueda binarios de la Figura 12.1 en el orden 2; 5; 5; 6; 7; 8. La corrección del algoritmo se sigue por inducción directamente de la propiedad del árbol de búsqueda binaria.

Lleva  $,n$  tiempo recorrer un árbol de búsqueda binaria de  $n$  nodos, ya que después de la llamada inicial, el procedimiento se llama a sí mismo recursivamente exactamente dos veces para cada nodo en el árbol: una para su hijo izquierdo y otra para su hijo derecho. El siguiente teorema da una prueba formal de que se necesita un tiempo lineal para realizar un recorrido de árbol en orden.

**Teorema 12.1 Si**

$x$  es la raíz de un subárbol de  $n$  nodos, entonces la llamada INORDER-TREE-WALK.x/ toma un tiempo  $,n$ .

**Demostración** Sea  $T .n$  el tiempo que tarda INORDER-TREE-WALK cuando se llama a la raíz de un subárbol de  $n$  nodos. Como INORDER-TREE-WALK visita todos los  $n$  nodos del subárbol, tenemos  $T .n / D .n$ . Queda por demostrar que  $T .n / D On$ .

Dado que INORDER-TREE-WALK toma una cantidad de tiempo pequeña y constante en un subárbol vacío (para la prueba  $x \neq \text{NIL}$ ), tenemos  $T .0 / D c$  para alguna constante  $c > 0$ .

Para  $n > 0$ , suponga que se llama a INORDER-TREE-WALK en un nodo  $x$  cuyo subárbol izquierdo tiene  $k$  nodos y cuyo subárbol derecho tiene  $n-k$  nodos. El tiempo para realizar INORDER-TREE-WALK.x/ está limitado por  $T .n / T .k / CT .nk1/Cd$  para alguna constante  $d > 0$  que refleja un límite superior en el tiempo para ejecutar el cuerpo de INORDER-TREE -WALK.x/, exclusivo del tiempo empleado en llamadas recursivas.

Usamos el método de sustitución para mostrar que  $T .n / D On$  demostrando que  $T .n / .c C d /n C c$ . Para  $n = 0$ , tenemos  $.c C d / 0C c = 0$ . Para  $n > 0$ , tenemos

$$\begin{aligned}
T .n &= T .k / CT .nk1 / C d \\
&= D ..c C d / k C c / C ..c C d / .nk1 / C c / C d D .c C d / n C c .c C d / \\
&\quad C c C d D .c C d / n c c
\end{aligned}$$

:

que completa la demostración. ■

## Ejercicios

### 12.1-1

Para el conjunto de f1; 4; 5; 10; dieciséis; 17; 21 g de claves, dibuje árboles de búsqueda binarios de alturas 2, 3, 4, 5 y 6.

### 12.1-2

¿Cuál es la diferencia entre la propiedad binary-search-tree y la propiedad min-heap (consulte la página 153)? ¿Se puede usar la propiedad min-heap para imprimir las claves de un árbol de n nodos en orden en On/time? Muestre cómo, o explique por qué no.

### 12.1-3

Proporcione un algoritmo no recursivo que realice un recorrido de árbol en orden. (Sugerencia: una solución fácil usa una pila como estructura de datos auxiliar. Una solución más complicada, pero elegante, no usa pila pero asume que podemos probar la igualdad de dos punteros).

### 12.1-4

Proporcione algoritmos recursivos que realicen recorridos de árboles de orden previo y posterior en tiempo .n/ en un árbol de n nodos.

### 12.1-5

Argumente que dado que ordenar n elementos toma  $.n \lg n/$  tiempo en el peor de los casos en el modelo de comparación, cualquier algoritmo basado en comparación para construir un árbol de búsqueda binario a partir de una lista arbitraria de n elementos toma  $.n \lg n/$  tiempo en lo peor caso.

---

## 12.2 Consultando un árbol de búsqueda binaria

A menudo necesitamos buscar una clave almacenada en un árbol de búsqueda binario. Además de la operación de BÚSQUEDA , los árboles de búsqueda binarios pueden admitir consultas como MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR. En esta sección examinaremos estas operaciones y mostraremos cómo respaldar cada una en el tiempo Oh/ en cualquier árbol binario de búsqueda de altura h.

buscando

Usamos el siguiente procedimiento para buscar un nodo con una clave dada en un árbol de búsqueda binaria. Dado un puntero a la raíz del árbol y una clave k, TREE-SEARCH devuelve un puntero a un nodo con clave k si existe; de lo contrario, devuelve NIL.

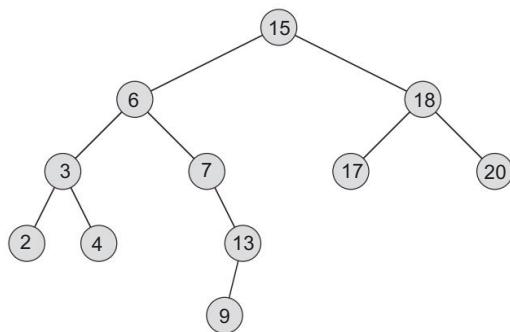


Figura 12.2 Consultas en un árbol de búsqueda binaria. Para buscar la clave 13 en el árbol, ¡seguimos el camino 15! 6! 7! 13 desde la raíz. La clave mínima en el árbol es 2, que se encuentra siguiendo los punteros izquierdos desde la raíz. La clave máxima 20 se encuentra siguiendo los punteros a la derecha desde la raíz. El sucesor del nodo con clave 15 es el nodo con clave 17, ya que es la clave mínima en el subárbol derecho de 15. El nodo con clave 13 no tiene subárbol derecho y, por lo tanto, su sucesor es su ancestro más bajo cuyo hijo izquierdo es también un antepasado. En este caso, el nodo con clave 15 es su sucesor.

```

ÁRBOL-BÚSQUEDA.x;
k/ 1 si x == NIL o k == x:key
      volver x
2 3 si k < x:key
      return TREE-SEARCH.x:left; k/ 4 5 else
      return TREE-SEARCH.x:right; k/
  
```

El procedimiento comienza su búsqueda en la raíz y traza un camino simple hacia abajo en el árbol, como se muestra en la Figura 12.2. Para cada nodo  $x$  que encuentra, compara la clave  $k$  con  $x:key$ . Si las dos claves son iguales, la búsqueda termina. Si  $k$  es menor que  $x:key$ , la búsqueda continúa en el subárbol izquierdo de  $x$ , ya que la propiedad del árbol de búsqueda binaria implica que  $k$  no se puede almacenar en el subárbol derecho. Simétricamente, si  $k$  es mayor que  $x:key$ , la búsqueda continúa en el subárbol derecho. Los nodos encontrados durante la recursión forman un camino simple hacia abajo desde la raíz del árbol y, por lo tanto, el tiempo de ejecución de TREE-SEARCH es  $O(h)$ , donde  $h$  es la altura del árbol.

Podemos reescribir este procedimiento de forma iterativa “desenrollando” la recursividad en un bucle while. En la mayoría de las computadoras, la versión iterativa es más eficiente.

**BÚSQUEDA-ITERATIVA-ÁRBOL.x; k/ 1**

```

mientras que x ≠ NIL y k ≠ x:tecla 2 si k
< x:tecla 3 x D x:izquierda
4 más x D x:derecha 5 volver
x

```

**mínimo y máximo**

Siempre podemos encontrar un elemento en un árbol de búsqueda binaria cuya clave sea un mínimo siguiendo los punteros secundarios izquierdos desde la raíz hasta que encontramos un NIL, como se muestra en la Figura 12.2. El siguiente procedimiento devuelve un puntero al elemento mínimo en el subárbol con raíz en un nodo dado x, que suponemos que no es NIL:

```

TREE-MINIMUM.x/ 1
while x:left ≠ NIL 2 x D
x:left 3 return x

```

La propiedad binary-search-tree garantiza que TREE-MINIMUM es correcto. Si un nodo x no tiene un subárbol izquierdo, entonces dado que cada clave en el subárbol derecho de x es al menos tan grande como x:clave, la clave mínima en el subárbol con raíz en x es x:clave. Si el nodo x tiene un subárbol izquierdo, entonces como ninguna clave en el subárbol derecho es más pequeña que x:clave y cada clave en el subárbol izquierdo no es mayor que x:clave, la clave mínima en el subárbol con raíz en x reside en el subárbol enraizado en x:izquierda.

El pseudocódigo para TREE-MAXIMUM es simétrico:

```

TREE-MAXIMUM.x/ 1
while x:right ≠ NIL 2 x D
x:right 3 return x

```

Ambos procedimientos se ejecutan en tiempo O(h) en un árbol de altura h ya que, como en la BÚSQUEDA DE ÁRBOL, la secuencia de nodos encontrados forma un camino simple hacia abajo desde la raíz.

**Sucesor y predecesor**

Dado un nodo en un árbol de búsqueda binaria, a veces necesitamos encontrar su sucesor en el orden determinado por un recorrido de árbol en orden. Si todas las claves son distintas, el

sucesor de un nodo  $x$  es el nodo con la clave más pequeña mayor que  $x$ :clave. La estructura de un árbol de búsqueda binaria nos permite determinar el sucesor de un nodo sin comparar claves. El siguiente procedimiento devuelve el sucesor de un nodo  $x$  en un árbol de búsqueda binaria si existe, y NIL si  $x$  tiene la clave más grande en el árbol:

```
TREE-SUCCESSOR.x/ 1
if x:right ≠ NIL 2 return
TREE-MINIMUM.x:right/ 3 y D x:p 4 while y ≠ NIL
and x == y:right
5 6 y D y:p 7 return y
x D y
```

Dividimos el código de TREE-SUCCESSOR en dos casos. Si el subárbol derecho del nodo  $x$  no está vacío, entonces el sucesor de  $x$  es solo el nodo más a la izquierda en el subárbol derecho de  $x$ , que encontramos en la línea 2 llamando a TREE-MINIMUM.x:right/. Por ejemplo, el sucesor del nodo con clave 15 en la Figura 12.2 es el nodo con clave 17.

Por otro lado, como le pide que muestre el ejercicio 12.2-6, si el subárbol derecho del nodo  $x$  está vacío y  $x$  tiene un sucesor  $y$ , entonces  $y$  es el antepasado más bajo de  $x$  cuyo hijo izquierdo también es un antepasado de  $x$ . En la Figura 12.2, el sucesor del nodo con clave 13 es el nodo con clave 15. Para encontrar  $y$ , simplemente subimos por el árbol desde  $x$  hasta que encontramos un nodo que es el hijo izquierdo de su padre; las líneas 3 a 7 de TREE-SUCCESSOR manejan este caso.

El tiempo de ejecución de TREE-SUCCESSOR en un árbol de altura  $h$  es  $O(h)$ , ya que o seguimos un camino simple hacia arriba del árbol o seguimos un camino simple hacia abajo del árbol. El procedimiento ÁRBOL-ANTERIOR, que es simétrico al ÁRBOL-SUCESOR, también se ejecuta en el tiempo  $O(h)$ .

Incluso si las claves no son distintas, definimos el sucesor y el predecesor de cualquier nodo  $x$  como el nodo devuelto por las llamadas realizadas a TREE-SUCCESSOR.x/ y TREE-PREDECESSOR.x/, respectivamente.

En resumen, hemos probado el siguiente teorema.

### Teorema 12.2

Podemos implementar las operaciones de conjunto dinámico BUSCAR, MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR para que cada uno se ejecute en tiempo  $O(h)$  en un árbol de búsqueda binaria de altura  $h$ . ■

## Ejercicios

### 12.2-1

Suponga que tenemos números entre 1 y 1000 en un árbol de búsqueda binaria y queremos buscar el número 363. ¿Cuál de las siguientes secuencias no podría ser la secuencia de nodos examinada?

- a. 2, 252, 401, 398, 330, 344, 397, 363.
- b. 924, 220, 911, 244, 898, 258, 362, 363.
- c. 925, 202, 911, 240, 912, 245, 363.
- d. 2, 399, 387, 219, 266, 382, 381, 278, 363.
- mi. 935, 278, 347, 621, 299, 392, 358, 363.

### 12.2-2

Escriba versiones recursivas de ÁRBOL-MÍNIMO y ÁRBOL-MÁXIMO.

### 12.2-3

Escriba el procedimiento ÁRBOL-ANTECESOR .

### 12.2-4

El profesor Bunyan cree haber descubierto una propiedad notable de los árboles binarios de búsqueda. Supongamos que la búsqueda de la clave  $k$  en un árbol de búsqueda binaria termina en una hoja. Considere tres conjuntos: A, las claves a la izquierda de la ruta de búsqueda; B, las claves en la ruta de búsqueda; y C, las teclas a la derecha de la ruta de búsqueda. El profesor Bunyan afirma que tres claves cualesquiera a 2 A, b 2 B y c 2 C deben satisfacer ab c. Dé un contraejemplo lo más pequeño posible a la afirmación del profesor.

### 12.2-5

Muestre que si un nodo en un árbol de búsqueda binario tiene dos hijos, entonces su sucesor no tiene hijo izquierdo y su predecesor no tiene hijo derecho.

### 12.2-6

Considere un árbol de búsqueda binario T cuyas claves son distintas. Demuestre que si el subárbol derecho de un nodo  $x$  en T está vacío y  $x$  tiene un sucesor  $y$ , entonces  $y$  es el ancestro más bajo de  $x$  cuyo hijo izquierdo también es un ancestro de  $x$ . (Recuerde que cada nodo es su propio ancestro).

### 12.2-7

Un método alternativo para realizar un recorrido de árbol en orden de un árbol de búsqueda binaria de  $n$  nodos encuentra el elemento mínimo en el árbol llamando a TREE-MINIMUM y luego haciendo  $n - 1$  llamadas a TREE-SUCCESSOR. Demuestre que este algoritmo se ejecuta en  $\sim n$  tiempo.

## 12.2-8

Demuestre que no importa en qué nodo comenzemos en un árbol de búsqueda binaria de altura h, k llamadas sucesivas a TREE-SUCCESSOR toman Ok C h/ tiempo.

## 12.2-9

Sea T un árbol de búsqueda binario cuyas claves son distintas, sea x un nodo hoja y sea y su padre. Muestre que y:key es la tecla más pequeña en T más grande que x:key o la tecla más grande en T más pequeña que x:key.

## 12.3 Inserción y eliminación

Las operaciones de inserción y eliminación hacen que cambie el conjunto dinámico representado por un árbol de búsqueda binaria. La estructura de datos debe modificarse para reflejar este cambio, pero de tal manera que la propiedad del árbol de búsqueda binaria se mantenga.

Como veremos, modificar el árbol para insertar un nuevo elemento es relativamente sencillo, pero manejar la eliminación es algo más complicado.

## Inserción

Para insertar un nuevo valor en un árbol binario de búsqueda T, usamos el procedimiento TREE-INSERT. El procedimiento toma un nodo ' para el cual ':key D , ':left D NIL, y ':right D NIL. Modifica T y algunos de los atributos de ' de tal manera que inserta ' en una posición apropiada en el árbol.

ÁRBOL-INSERCIÓN.T; '

```

1 y D NIL 2 x
D T:root 3 while x
  ▷ NIL 4 y D x 5 if ':key
    < x:key 6 x D x:left
    7 else x D x:right 8 ':p D y 9 if
      y == CERO 10

```

```

T:root D ' 11           // el árbol T estaba vacío
elseif ':key < y:key 12 y:left D
' 13 else y:right D '

```

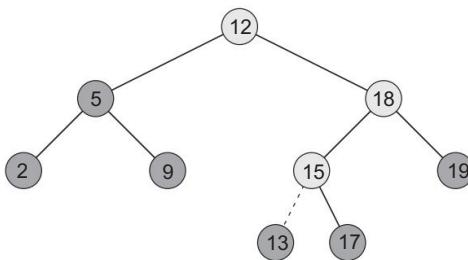


Figura 12.3 Inserción de un elemento con clave 13 en un árbol de búsqueda binaria. Los nodos ligeramente sombreados indican la ruta simple desde la raíz hasta la posición donde se inserta el elemento. La línea discontinua indica el enlace en el árbol que se agrega para insertar el elemento.

La figura 12.3 muestra cómo funciona TREE-INSERT . Al igual que los procedimientos TREE SEARCH y ITERATIVE-TREE-SEARCH, TREE-INSERT comienza en la raíz del árbol y el puntero x traza un camino simple hacia abajo buscando un NIL para reemplazarlo con el elemento de entrada ' . El procedimiento mantiene el puntero final y como padre de x . Después de la inicialización, el bucle while de las líneas 3 a 7 hace que estos dos punteros se muevan hacia abajo en el árbol, yendo hacia la izquierda o hacia la derecha, dependiendo de la comparación de ':key con x:key , hasta que x se convierte en NIL . Este NIL ocupa la posición donde deseamos colocar el elemento de entrada ' . Necesitamos el puntero final y , porque cuando encontramos el NIL al que pertenece ' , la búsqueda ha avanzado un paso más allá del nodo que debe cambiarse. Las líneas 8 a 13 establecen los punteros que hacen que se inserte ' .

Como las otras operaciones primitivas en los árboles de búsqueda, el procedimiento TREE-INSERT corre en  $O(h)$  tiempo sobre un árbol de altura h .

### Supresión

La estrategia general para eliminar un nodo ' de un árbol de búsqueda binario T tiene tres casos básicos pero, como veremos, uno de los casos es un poco engañoso.

Si ' no tiene hijos, entonces simplemente lo eliminamos modificando su padre para reemplazar ' con NIL como su hijo.

Si ' tiene solo un hijo, entonces elevamos ese hijo para que tome la posición de ' en el árbol modificando el padre de ' para reemplazar ' por el hijo de ' .

Si ' tiene dos hijos, entonces encontramos el sucesor y de ' , que debe estar en el subárbol derecho de ' , y hacemos que y tome la posición de ' en el árbol. El resto del subárbol derecho original de ' se convierte en el nuevo subárbol derecho de y , y el subárbol izquierdo de ' se convierte en el nuevo subárbol izquierdo de y . Este caso es complicado porque, como veremos, importa si y es el hijo derecho de ' .

El procedimiento para eliminar un nodo dado  $\alpha$  de un árbol binario de búsqueda  $T$  toma como argumentos los punteros a  $T$  y  $\alpha$ . Organiza sus casos de manera un poco diferente a los tres casos descritos anteriormente al considerar los cuatro casos que se muestran en la Figura 12.4.

Si  $\alpha$  no tiene hijo izquierdo (parte (a) de la figura), entonces reemplazamos  $\alpha$  por su hijo derecho, que puede ser NIL o no. Cuando el hijo derecho de  $\alpha$  es NIL, este caso se trata de la situación en la que  $\alpha$  no tiene hijos. Cuando el hijo derecho de  $\alpha$  no es NIL, este caso maneja la situación en la que  $\alpha$  tiene solo un hijo, que es su hijo derecho.

Si  $\alpha$  tiene solo un hijo, que es su hijo izquierdo (parte (b) de la figura), entonces reemplazamos  $\alpha$  por su hijo izquierdo.

De lo contrario,  $\alpha$  tiene un hijo izquierdo y uno derecho. Encontramos el sucesor  $y$  de  $\alpha$ , que se encuentra en el subárbol derecho de  $\alpha$  y no tiene hijo izquierdo (vea el ejercicio 12.2-5).

Queremos separar  $y$  de su ubicación actual y hacer que reemplace  $\alpha$  en el árbol.

Si  $y$  es el hijo derecho de  $\alpha$  (parte (c)), entonces reemplazamos  $\alpha$  por  $y$ , dejando solo al hijo derecho de  $y$ .

De lo contrario,  $y$  se encuentra dentro del subárbol derecho de  $\alpha$  pero no es el hijo derecho de  $\alpha$  (parte (d)).

En este caso, primero reemplazamos  $y$  por su propio hijo y luego reemplazamos  $\alpha$  por  $y$ .

Para mover los subárboles dentro del árbol de búsqueda binaria, definimos una subrutina TRANSPLANT, que reemplaza un subárbol como hijo de su padre con otro subárbol. Cuando TRANSPLANT reemplaza el subárbol enraizado en el nodo  $u$  con el subárbol enraizado en el nodo  $v$ , el padre del nodo  $u$  se convierte en el padre del nodo  $v$ , y el padre de  $u$  termina teniendo como su hijo apropiado.

TRANSPLANTE( $T; u; v$ )

```

u:p == NIL 2 T:root
D 3 elseif u ==
u:p:left 4 u:p:left D 5 else
u:p:right D 6 if ≠ NIL 7

```

:p D tu:p

Las líneas 1 y 2 manejan el caso en el que  $u$  es la raíz de  $T$ . De lo contrario,  $u$  es un hijo izquierdo o un hijo derecho de su padre. Las líneas 3 y 4 se encargan de actualizar  $u:p:left$  si  $u$  es un hijo izquierdo, y la línea 5 actualiza  $u:p:right$  si  $u$  es un hijo derecho. Permitimos que sea NIL, y las líneas 6–7 actualizan :p si no es NIL. Tenga en cuenta que TRANSPLANT no intenta actualizar :left y :right; hacerlo, o no hacerlo, es responsabilidad de la persona que llama a TRANSPLANT .

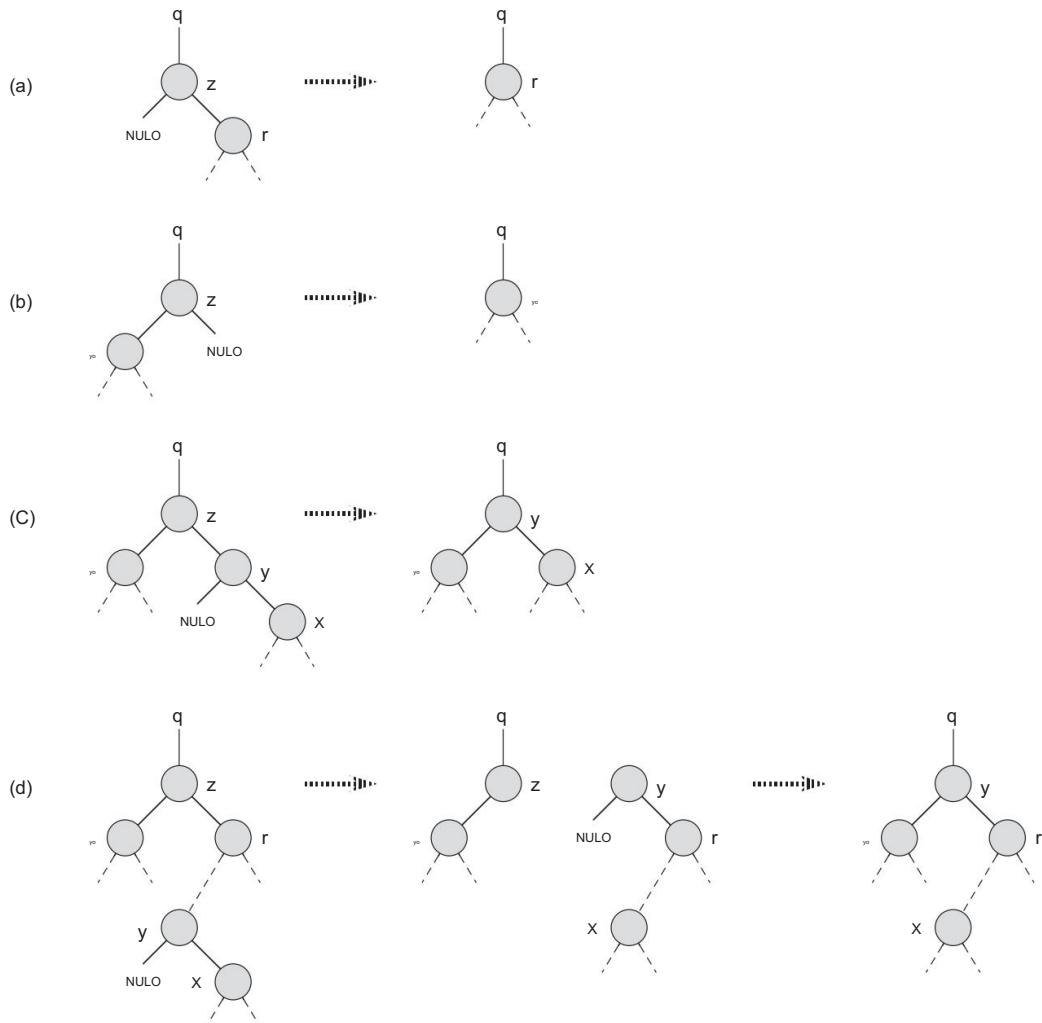


Figura 12.4 Eliminación de un nodo 'l' de un árbol de búsqueda binaria. El nodo 'l' puede ser la raíz, un hijo izquierdo del nodo q, o un hijo derecho de q. (a) El nodo 'l' no tiene hijo izquierdo. Reemplazamos 'l' por su hijo derecho r, que puede o no ser NIL. (b) El nodo 'l' tiene un hijo izquierdo l pero ningún hijo derecho. Sustituimos 'l' por l. (c) Nodo 'l' tiene dos hijos; su hijo izquierdo es el nodo l, su hijo derecho es su sucesor y, y el hijo derecho de y es el nodo x. Reemplazamos 'l' por y, actualizando el hijo izquierdo de y para convertirlo en l, pero dejando x como el hijo derecho de y. (d) El nodo 'l' tiene dos hijos (el hijo izquierdo l y el hijo derecho r), y su sucesor y ≠ r se encuentra dentro del subárbol con raíz en r. Reemplazamos y por su propio hijo x, y hacemos que y sea el padre de r. Entonces, hacemos que y sea el hijo de q y el padre de l.

Con el procedimiento de TRASPLANTE en mano, aquí está el procedimiento que elimina nodo ' del árbol de búsqueda binaria T :

```
ÁRBOL-BORRAR.T; '/ 1
si ':izquierda == NIL
    TRASPLANTE.T; ':right/ 2 3
elseif ':right == NIL 4
    TRANSPLANT.T; ':left/ 5 else y D TREE-
MINIMUM.':right/ 6 if y:p ≠ ' 7 TRANSPLANT.T;
y; y:derecha/ 8
y:derecha D ':derecha 9 y:derecha:p D y 10
TRANSPLANTE.T; y/ 11 y:izquierda
D ':izquierda 12 y:izquierda:p D y
```

El procedimiento TREE-DELETE ejecuta los cuatro casos de la siguiente manera. Las líneas 1 y 2 manejan el caso en el que el nodo ' no tiene un hijo izquierdo, y las líneas 3 y 4 manejan el caso en el que ' tiene un hijo izquierdo pero no un hijo derecho. Las líneas 5 a 12 tratan de los dos casos restantes, en los que ' tiene dos hijos. La línea 5 encuentra el nodo y, que es el sucesor de '. Debido a que ' tiene un subárbol derecho no vacío, su sucesor debe ser el nodo en ese subárbol con la clave más pequeña; de ahí la llamada a TREE-MINIMUM.':right/. Como notamos antes, y no tiene un hijo izquierdo. Queremos separar y de su ubicación actual, y debería reemplazar a ' en el árbol. Si y es el hijo derecho de ', entonces las líneas 10–12 reemplazan ' como hijo de su padre por y y reemplazan el hijo izquierdo de y por el hijo izquierdo de '. Si y no es el hijo izquierdo de ', las líneas 7–9 reemplazan y como hijo de su padre por el hijo derecho de y y convierten el hijo derecho de ' en el hijo derecho de y, y luego las líneas 10–12 reemplazan ' como hijo de su padre por y y reemplaza el hijo izquierdo de y por el hijo izquierdo de '.

Cada línea de TREE-DELETE, incluidas las llamadas a TRANSPLANT, toma un tiempo constante, excepto la llamada a TREE-MINIMUM en la línea 5. Así, TREE-DELETE se ejecuta en tiempo O(h) en un árbol de altura h.

En resumen, hemos probado el siguiente teorema.

### Teorema 12.3

Podemos implementar las operaciones de conjunto dinámico INSERT y DELETE para que cada una se ejecute en tiempo O(h) en un árbol de búsqueda binaria de altura h. ■

### Ejercicios

#### 12.3-1

Proporcione una versión recursiva del procedimiento TREE-INSERT .

#### 12.3-2

Supongamos que construimos un árbol de búsqueda binaria insertando repetidamente valores distintos en el árbol. Argumente que el número de nodos examinados al buscar un valor en el árbol es uno más el número de nodos examinados cuando el valor se insertó por primera vez en el árbol.

#### 12.3-3

Podemos ordenar un conjunto dado de n números construyendo primero un árbol de búsqueda binaria que contenga estos números (usando TREE-INSERT repetidamente para insertar los números uno por uno) y luego imprimiendo los números siguiendo un recorrido de árbol en orden. ¿Cuáles son los tiempos de ejecución en el peor de los casos y en el mejor de los casos para este algoritmo de clasificación?

#### 12.3-4

¿La operación de eliminación es "conmutativa" en el sentido de que eliminar x y luego y de un árbol de búsqueda binaria deja el mismo árbol que eliminar y y luego x? Argumente por qué es así o dé un contraejemplo.

#### 12.3-5

Suponga que en lugar de que cada nodo x mantenga el atributo x:p, apuntando al padre de x, mantiene x:succ, apuntando al sucesor de x. Proporcione un pseudocódigo para BUSCAR, INSERTAR y ELIMINAR en un árbol de búsqueda binario T usando esta representación. Estos procedimientos deben operar en el tiempo Oh/, donde h es la altura del árbol T . (Sugerencia: es posible que desee implementar una subrutina que devuelva el parent de un nodo).

#### 12.3-6

Cuando el nodo ' en TREE-DELETE tiene dos hijos, podemos elegir el nodo y como su predecesor en lugar de su sucesor. ¿Qué otros cambios en TREE-DELETE serían necesarios si lo hicieramos? Algunos han argumentado que una estrategia justa, que otorga la misma prioridad al predecesor y al sucesor, produce un mejor desempeño empírico.

¿Cómo se podría cambiar TREE-DELETE para implementar una estrategia tan justa?

---

## ? 12.4 Árboles de búsqueda binarios construidos aleatoriamente

Hemos demostrado que cada una de las operaciones básicas en un árbol de búsqueda binaria se ejecuta en tiempo Oh/, donde h es la altura del árbol. El colmo de una búsqueda binaria

Sin embargo, el árbol varía a medida que se insertan y eliminan elementos. Si, por ejemplo, los  $n$  elementos se insertan en orden estrictamente creciente, el árbol será una cadena con altura  $n - 1$ .

Por otro lado, el ejercicio B.5-4 muestra que  $h \leq \lg n$ . Al igual que con quicksort, podemos mostrar que el comportamiento del caso promedio está mucho más cerca del mejor caso que del peor caso.

Desafortunadamente, se sabe poco sobre la altura promedio de un árbol de búsqueda binaria cuando se utilizan tanto la inserción como la eliminación para crearlo. Cuando se crea el árbol solo con la inserción, el análisis se vuelve más manejable. Por lo tanto, definamos un árbol de búsqueda binario construido aleatoriamente en  $n$  claves como uno que surge de insertar las claves en orden aleatorio en un árbol inicialmente vacío, donde cada una de las  $n!/\sqrt{n}$  permutaciones de las claves de entrada es igualmente probable. (El ejercicio 12.4-3 le pide que demuestre que esta noción es diferente de asumir que todos los árboles de búsqueda binarios en  $n$  claves son igualmente probables). En esta sección demostraremos el siguiente teorema.

#### Teorema 12.4 La

altura esperada de un árbol binario de búsqueda construido aleatoriamente sobre  $n$  claves distintas es  $O(\lg n)$ .

Prueba Comenzamos definiendo tres variables aleatorias que ayudan a medir la altura de un árbol de búsqueda binario construido aleatoriamente. Denotamos la altura de una búsqueda binaria construida aleatoriamente en  $n$  claves por  $X_n$ , y definimos la altura exponencial  $Y_n = 2^{X_n}$ . Cuando construimos un árbol de búsqueda binaria en  $n$  claves, elegimos una clave como la de la raíz, y hacemos que  $R_n$  denote la variable aleatoria que ocupa el rango de esta clave dentro del conjunto de  $n$  claves; es decir,  $R_n$  ocupa la posición que ocuparía esta clave si el conjunto de claves estuviera ordenado. Es igualmente probable que el valor de  $R_n$  sea cualquier elemento del conjunto  $\{1, 2, \dots, n\}$ . Si  $R_n = i$ , entonces el subárbol izquierdo de la raíz es un árbol de búsqueda binario construido aleatoriamente en claves  $1, 2, \dots, i$ , y el subárbol derecho es un árbol de búsqueda binario construido aleatoriamente en claves  $i+1, \dots, n$ . Debido a que la altura de un árbol binario es 1 más que la mayor de las alturas de los dos subárboles de la raíz, la altura exponencial de un árbol binario es el doble de la altura exponencial de los dos subárboles de la raíz. Si sabemos que  $R_n = i$ , se sigue que

$$Y_n = 2^{\max(Y_1, Y_2, \dots, Y_i)}$$

Como casos base tenemos que  $Y_1 = 1$ , porque la altura exponencial de un árbol con 1 nodo es 2 y, por conveniencia, definimos  $Y_0 = 0$ .

A continuación, defina las variables aleatorias indicadoras  $Z_{n,1}, Z_{n,2}, \dots, Z_{n,n}$ , donde

$Z_{n,i} = 1 \text{ si } R_n = i$  : Porque

$R_n$  es igualmente probable que sea cualquier elemento de  $\{1, 2, \dots, n\}$ , se sigue que  $\Pr[R_n = i] = 1/n$  para  $i = 1, 2, \dots, n$ , y por lo tanto, por el Lema 5.1, tenemos  $E[Z_{n,i}] = 1/n$  ;

(12.1)

para  $i \in D_1; 2; \dots; n$ : norte. Como exactamente un valor de  $Z_{n;i}$  es 1 y todos los demás son 0, también tenemos

$$\sum_{i \in D_1} Y_n D X_n = Z_{n;i} \cdot 2^{\max(Y_1; Y_n)} : \quad (12.1)$$

Mostraremos que  $E \sum_{i \in D_1} Y_n$  es un polinomio en  $n$ , lo que finalmente implicará que  $E \sum_{i \in D_1} D O \lg n$ .

Afirmamos que la variable aleatoria indicadora  $Z_{n;i}$  es independiente de los valores de  $Y_1$  e  $Y_n$ . Habiendo elegido  $R_n \in D_1$ , el subárbol izquierdo (cuya altura exponencial es  $Y_1$ ) se construye aleatoriamente sobre las claves  $i < 1$  cuyos rangos son menores que  $i$ . Este subárbol es como cualquier otro árbol de búsqueda binario creado aleatoriamente en las claves  $i < 1$ . Aparte del número de claves que contiene, la estructura de este subárbol no se ve afectada en absoluto por la elección de  $R_n \in D_1$  y, por lo tanto, las variables aleatorias  $Y_1$  y  $Z_{n;i}$  son independientes.

Asimismo, el subárbol derecho, cuya altura exponencial es  $Y_n$ , se construye aleatoriamente sobre las claves  $i > n$  cuyos rangos son mayores que  $i$ .

Su estructura es independiente del valor de  $R_n$ , por lo que las variables aleatorias  $Y_n$  y  $Z_{n;i}$  son independientes. Por lo tanto, tenemos

$$E \sum_{i \in D_1} Y_n = E \sum_{i \in D_1} Z_{n;i} \cdot 2^{\max(Y_1; Y_n)}$$

$$= \sum_{i \in D_1} E Z_{n;i} \cdot 2^{\max(Y_1; Y_n)} : \quad (\text{por linealidad de la expectativa})$$

$$= \sum_{i \in D_1} E Z_{n;i} \cdot 2^{\max(Y_1; Y_n)} : \quad (\text{por independencia})$$

$$= \sum_{i \in D_1} \frac{1}{2^n} E 2^{\max(Y_1; Y_n)} : \quad (\text{por la ecuación (12.1)})$$

$$= \sum_{i \in D_1} \frac{2}{2^n} E \max(Y_1; Y_n) : \quad (\text{por ecuación (C.22)})$$

$$= \sum_{i \in D_1} \frac{2}{2^n} E Y_1 + E Y_n : \quad (\text{por el ejercicio C.3-4}).$$

Como cada término  $E \sum_{i \in D_1} Y_n$ ;  $E \sum_{i \in D_2} Y_n$ ; ...;  $E \sum_{i \in D_n} Y_n$  aparece dos veces en la última suma, una como  $E \sum_{i \in D_1} Y_n$  y otra como  $E \sum_{i \in D_n} Y_n$ , tenemos la recurrencia

$$E \sum_{i \in D_n} Y_n = \frac{4}{2^n} \sum_{i \in D_0} E Y_i : \quad (12.2)$$

Usando el método de sustitución, mostraremos que para todos los enteros positivos  $n$ , el recurrencia (12.2) tiene la solución

$$\frac{1}{4} n C_3 \\ 3! :$$

Al hacerlo, utilizaremos la identidad

$$\begin{array}{ccc} & \text{yo C 3} & n C 3 \\ Xn1 & & \\ iD0 & 3! D & 4! : \end{array} \quad (12.3)$$

(El ejercicio 12.4-1 le pide que pruebe esta identidad).

Para los casos base, notamos que los límites 0 D Y0 DE  $\Omega Y_0 .1=4/3$       3 D 1=4

y 1 D Y1 DE  $\Omega Y_1 .1=4/1C3$       3 D 1 aguantado. Para el caso inductivo, tenemos que

$$\begin{aligned} E \Omega Y_n &= \frac{4}{\text{norte}} Xn1 E \Omega Y_i \\ &= \frac{4}{\text{norte}} \frac{1}{iD0} \text{yo C 3} \\ &\quad 3! (\text{por la hipótesis inductiva}) 3 ! 4! \\ D &= \frac{1}{\text{norte}} \text{yo C 3} \\ &= \frac{1}{\text{norte}} \frac{n C 3}{iD0} \quad (\text{por la ecuación (12.3)}) \\ D &= \frac{1}{\text{norte}} \frac{\dots n C 3 / \check{S}}{4 \check{S} \dots n 1 / \check{S} \dots n} \\ D &= \frac{1}{4} \frac{C 3 / \check{S}}{3 \check{S} n \check{S}} \\ D &= \frac{1}{4} n C 3 \\ &\quad 3! : \end{aligned}$$

Hemos acotado  $E \Omega Y_n$ , pero nuestro objetivo final es acotar  $E \Omega X_n$ . Como se le pide que muestre en el ejercicio 12.4-4, la función  $f .x / D 2x$  es convexa (vea la página 1199).

Por lo tanto, podemos emplear la desigualdad de Jensen (C.26), que dice que

$2E \Omega X_n \leq 2X_n$

$\leq DE \Omega Y_n ;$

como sigue:

$$\begin{array}{ccc} 2E \Omega X_n & \frac{1}{4} n C 3 \\ & 3! \end{array}$$

$$\begin{array}{r} 1 \\ \times n C 3 / .n C 2 / .n C 1 / 4 6 n 3 \\ \hline C 6 n^2 C 11 n C 6 \end{array}$$

$$\begin{array}{r} \\ \\ \hline 24 \end{array}$$

Tomando logaritmos de ambos lados se obtiene  $E \propto \ln D O(\lg n)$ . ■

### Ejercicios

12.4-1

Demostrar la ecuación (12.3).

12.4-2

Describa un árbol de búsqueda binario en  $n$  nodos tal que la profundidad promedio de un nodo en el árbol sea  $\lg n$  pero la altura del árbol sea  $\lceil \lg n \rceil$ . Proporcione un límite superior asintótico para la altura de un árbol de búsqueda binaria de  $n$  nodos en el que la profundidad promedio de un nodo sea  $\lg n$ .

12.4-3

Muestre que la noción de un árbol de búsqueda binario elegido al azar en  $n$  claves, donde cada árbol de búsqueda binario de  $n$  claves tiene la misma probabilidad de ser elegido, es diferente de la noción de un árbol de búsqueda binario construido aleatoriamente que se da en esta sección. (Sugerencia: Haga una lista de las posibilidades cuando  $n=3$ .)

12.4-4

Muestre que la función  $f(x) = 2x$  es convexa.

12.4-5 ?

Considere RANDOMIZED-QUICKSORT operando en una secuencia de  $n$  números de entrada distintos. Demuestre que para cualquier constante  $k > 0$ , todas menos  $O(1/nk)$  de las  $n!$  permutaciones de entrada producen un tiempo de ejecución  $O(n \lg n)$ .

### Problemas

12-1 Árboles de búsqueda binarios con claves iguales

Las claves iguales plantean un problema para la implementación de árboles de búsqueda binarios.

- a. ¿Cuál es el rendimiento asintótico de TREE-INSERT cuando se usa para insertar  $n$  elementos con claves idénticas en un árbol de búsqueda binario inicialmente vacío?

Proponemos mejorar TREE-INSERT probando antes de la línea 5 para determinar si  $'key' < key$  y probando antes de la línea 11 para determinar si  $'key' > key$ .

Si se mantiene la igualdad, implementamos una de las siguientes estrategias. Para cada estrategia, encuentre el rendimiento asintótico de insertar  $n$  elementos con claves idénticas en un árbol de búsqueda binario inicialmente vacío. (Las estrategias se describen para la línea 5, en la que comparamos las claves de  $'$  y  $x$ . Sustituya  $y$  por  $x$  para llegar a las estrategias de la línea 11).

- b. Mantenga un indicador booleano  $x:b$  en el nodo  $x$  y establezca  $x$  en  $x:\text{izquierda}$  o  $x:\text{derecha}$  según el valor de  $x:b$ , que alterna entre FALSO y VERDADERO cada vez que visitamos  $x$  al insertar un nodo con el mismo clave como  $x$ .
- C. Mantenga una lista de nodos con claves iguales en  $x$  e inserte  $'$  en la lista.
- d. Establezca aleatoriamente  $x$  en  $x:\text{izquierda}$  o  $x:\text{derecha}$ . (Proporcione el rendimiento en el peor de los casos y obtenga informalmente el tiempo de ejecución esperado).

### 12-2 árboles Radix

Dadas dos cadenas  $a D a_0a_1 \dots a_p$  y  $b D b_0b_1 \dots b_q$ , donde cada  $a_i$  y cada  $b_j$  están en algún conjunto ordenado de caracteres, decimos que la cadena  $a$  es lexicográficamente menor que la cadena  $b$  si

1. existe un entero  $j$ , donde  $0 \leq j \leq \min(p, q)$ , tal que  $a_i < b_i$  para todo  $i \in \{0, 1, \dots, j\}$  y  $a_j < b_j$ ,

pag.

Por ejemplo, si  $a$  y  $b$  son cadenas de bits, entonces  $10100 < 10110$  por la regla 1 (dejando a  $j = 3$ ) y  $10100 < 101000$  por la regla 2. Este orden es similar al que se usa en los diccionarios del idioma inglés.

La estructura de datos del árbol radix que se muestra en la figura 12.5 almacena las cadenas de bits 1011, 10, 011, 100 y 0. Al buscar una clave  $a D a_0a_1 \dots a_p$ , vamos a la izquierda en un nodo de profundidad  $i$  si  $a_i \leq 0$  ya la derecha si  $a_i \geq 1$ . Sea  $S$  un conjunto de cadenas de bits distintas cuyas longitudes suman  $n$ . Muestre cómo usar un árbol radix para clasificar  $S$  lexicográficamente en tiempo  $\sim n$ . Para el ejemplo de la figura 12.5, la salida de la clasificación debe ser la secuencia 0, 011, 10, 100, 1011.

### 12-3 Profundidad promedio de un nodo en un árbol de búsqueda binario

construido aleatoriamente En este problema, demostramos que la profundidad promedio de un nodo en un árbol de búsqueda binario construido aleatoriamente con  $n$  nodos es  $O(\lg n)$ . Aunque este resultado es más débil que el del Teorema 12.4, la técnica que usaremos revela una similitud sorprendente entre la construcción de un árbol de búsqueda binaria y la ejecución de RANDOMIZED QUICKSORT de la Sección 7.3.

Definimos la longitud total del camino  $P(T)$  de un árbol binario  $T$  como la suma, sobre todos los nodos  $x$  en  $T$ , de la profundidad del nodo  $x$ , que denotamos por  $dx(T)$ .

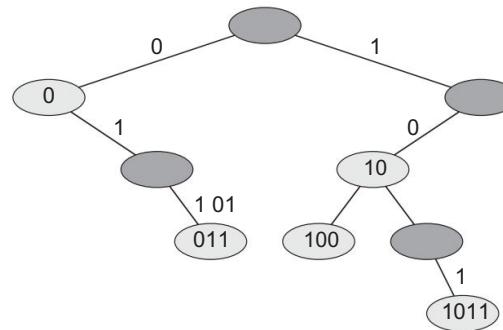


Figura 12.5 Un árbol de base que almacena las cadenas de bits 1011, 10, 011, 100 y 0. Podemos determinar la clave de cada nodo recorriendo la ruta simple desde la raíz hasta ese nodo. No hay necesidad, por tanto, de almacenar las claves en los nodos; las claves aparecen aquí solo con fines ilustrativos. Los nodos están muy sombreados si las claves correspondientes a ellos no están en el árbol; tales nodos están presentes solo para establecer una ruta a otros nodos.

a. Argumente que la profundidad promedio de un nodo en T es

$$\frac{1}{\text{#nodos}} \sum_{\text{nodo}} \text{P}(\text{nodo}) \cdot \text{D}(T)$$

Por lo tanto, deseamos mostrar que el valor esperado de  $\text{P}(\text{nodo}) \cdot \text{D}(T)$  es  $O(n \lg n)$ .

b. Sean TL y TR los subárboles izquierdo y derecho del árbol T, respectivamente. Argumente que si T tiene n nodos, entonces

$$\text{P}(\text{nodo}) \cdot \text{D}(T) = \text{P}(\text{nodo}) \cdot \text{D}(TL) + \text{P}(\text{nodo}) \cdot \text{D}(TR)$$

c. Sea P(n) la longitud de ruta total promedio de una búsqueda binaria construida aleatoriamente árbol con n nodos. Muestra esa

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} i \cdot \text{P}(i)$$

d. Muestre cómo reescribir P(n) como

$$P(n) = \frac{2}{n} \sum_{k=1}^{\lfloor n/2 \rfloor} k \cdot \text{P}(k)$$

mi. Recordando el análisis alternativo de la versión aleatoria de ordenación rápida dada en el problema 7-3, concluya que  $P(n) = O(n \lg n)$ .

En cada invocación recursiva de ordenación rápida, elegimos un elemento de pivote aleatorio para dividir el conjunto de elementos que se ordenan. Cada nodo de un árbol de búsqueda binaria divide el conjunto de elementos que caen en el subárbol enraizado en ese nodo.

- F. Describa una implementación de clasificación rápida en la que las comparaciones para clasificar un conjunto de elementos son exactamente las mismas que las comparaciones para insertar los elementos en un árbol de búsqueda binaria. (El orden en que se hacen las comparaciones puede diferir, pero deben ocurrir las mismas comparaciones).

#### 12-4 Número de árboles binarios diferentes

Sea  $b_n$  el número de árboles binarios diferentes con  $n$  nodos. En este problema, encontrarás una fórmula para  $b_n$ , así como una estimación asintótica.

- a. Demuestre que  $b_0 = 1$  y que, para  $n \geq 1$ ,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}$$

- b. Con referencia al problema 4-4 para la definición de una función generadora, sea  $B(x)$  ser la función generadora

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

Muestre que  $B(x) = xB(x)/2 + 1$ , y por lo tanto una forma de expresar  $B(x)$  es

$$B(x) = \frac{1}{2x} (1 + \sqrt{1 + 4x})$$

La expansión de Taylor de  $f(x)$  alrededor del punto  $x = a$  viene dada por

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x-a)^k; \text{ Kansas}$$

donde  $f^{(k)}(x)$  es la  $k$ -ésima derivada de  $f$  evaluada en  $x$ .

- C. Muestra esa

$$b_n = \frac{1}{n!} \binom{2n}{n}$$

(el enésimo número catalán) usando la expansión de Taylor de  $p_1(4x)$  alrededor de  $x=0$ . (Si lo desea, en lugar de usar la expansión de Taylor, puede usar la generalización de la expansión binomial (C.4) a exponentes no integrales  $n$ , donde para cualquier número real  $n$  y para cualquier entero  $k$ , interpretamos que es  $\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k!}$  si  $k \geq 0$ , y 0 en caso contrario.)

nk

d. Muestra esa

$$\binom{4n}{n} = \frac{(4n)!}{n!(n+1)!} : p(n) = 2^n$$

## Notas del capítulo

Knuth [211] contiene una buena discusión de árboles de búsqueda binarios simples, así como muchas variaciones. Los árboles de búsqueda binarios parecen haber sido descubiertos de forma independiente por varias personas a fines de la década de 1950. Los árboles Radix a menudo se denominan "intentos", que provienen de las letras del medio en la recuperación de la palabra. Knuth [211] también los analiza.

Muchos textos, incluidas las dos primeras ediciones de este libro, tienen un método algo más simple para eliminar un nodo de un árbol de búsqueda binaria cuando sus dos hijos están presentes. En lugar de reemplazar el nodo  $x$  por su sucesor  $y$ , eliminamos el nodo y pero copiamos su clave y los datos del satélite en el nodo  $y$ . La desventaja de este enfoque es que el nodo realmente eliminado podría no ser el nodo pasado al procedimiento de eliminación. Si otros componentes de un programa mantienen punteros a los nodos en el árbol, podrían terminar por error con punteros "obsoletos" a los nodos que se han eliminado. Aunque el método de eliminación presentado en esta edición de este libro es un poco más complicado, garantiza que una llamada para eliminar nodo  $x$  elimina nodo  $x$  y solo nodo  $x$ .

La sección 15.5 mostrará cómo construir un árbol de búsqueda binario óptimo cuando conocemos las frecuencias de búsqueda antes de construir el árbol. Es decir, dadas las frecuencias de búsqueda de cada clave y las frecuencias de búsqueda de valores que caen entre las claves del árbol, construimos un árbol de búsqueda binaria para el cual un conjunto de búsquedas que sigue a estas frecuencias examina el número mínimo de nodos.

La prueba de la Sección 12.4 que limita la altura esperada de un árbol de búsqueda binario construido aleatoriamente se debe a Aslam [24]. Martínez y Roura [243] dan algoritmos aleatorios para la inserción y eliminación de árboles de búsqueda binarios en los que el resultado de cualquiera de las dos operaciones es un árbol de búsqueda binario aleatorio. Sin embargo, su definición de un árbol de búsqueda binario aleatorio difiere, solo ligeramente, de la de un árbol de búsqueda binario construido aleatoriamente en este capítulo.

---

## 13

## Árboles rojos y negros

El capítulo 12 mostró que un árbol de búsqueda binaria de altura  $h$  puede admitir cualquiera de las operaciones básicas de conjuntos dinámicos, como BUSCAR, PREDECESOR, SUCESOR, MÍNIMO, MÁXIMO, INSERTAR y ELIMINAR, en tiempo  $O(h)$ . Así, las operaciones de conjunto son rápidas si la altura del árbol de búsqueda es pequeña. Sin embargo, si su altura es grande, es posible que las operaciones de conjunto no se ejecuten más rápido que con una lista enlazada. Los árboles rojo-negro son uno de los muchos esquemas de árboles de búsqueda que están "equilibrados" para garantizar que las operaciones básicas de conjuntos dinámicos tarden  $O(\lg n)$  en el peor de los casos.

---

### 13.1 Propiedades de los árboles rojo-negro

Un árbol rojo-negro es un árbol de búsqueda binaria con un bit adicional de almacenamiento por nodo: su color, que puede ser ROJO o NEGRO. Al restringir los colores de los nodos en cualquier camino simple desde la raíz hasta una hoja, los árboles rojo-negro aseguran que ningún camino sea más del doble de largo que cualquier otro, de modo que el árbol esté aproximadamente equilibrado .

Cada nodo del árbol ahora contiene los atributos color, clave, izquierda, derecha y p. Si no existe un elemento secundario o principal de un nodo, el atributo de puntero correspondiente del nodo contiene el valor NIL. Consideraremos estos NIL como punteros a hojas (nodos externos) del árbol de búsqueda binaria y los nodos normales portadores de claves como nodos internos del árbol.

Un árbol rojo-negro es un árbol binario que satisface las siguientes propiedades rojo-negro:

1. Cada nodo es rojo o negro.
2. La raíz es negra.
3. Cada hoja (NIL) es negra.
4. Si un nodo es rojo, entonces sus dos hijos son negros.
5. Para cada nodo, todos los caminos simples desde el nodo hasta las hojas descendientes contienen el mismo número de nodos negros.

La Figura 13.1(a) muestra un ejemplo de un árbol rojo-negro.

Por conveniencia al tratar con las condiciones de contorno en el código de árbol rojo-negro, usamos un solo centinela para representar NIL (consulte la página 238). Para un árbol T rojo-negro, el centinela T:nil es un objeto con los mismos atributos que un nodo ordinario en el árbol. Su atributo de color es NEGRO, y sus otros atributos (p, izquierda, derecha y clave) pueden tomar valores arbitrarios. Como muestra la figura 13.1(b), todos los punteros a NIL se reemplazan por punteros al centinela T:nil.

Usamos el centinela para que podamos tratar un hijo NIL de un nodo x como un nodo ordinario cuyo padre es x. Aunque en su lugar podríamos agregar un nodo centinela distinto para cada NIL en el árbol, de modo que el padre de cada NIL esté bien definido, ese enfoque desperdiciaría espacio. En su lugar, usamos el único centinela T:nil para representar todos los NIL: todas las hojas y el padre de la raíz. Los valores de los atributos p, izquierda, derecha y clave del centinela son irrelevantes, aunque podemos establecerlos durante el curso de un procedimiento para nuestra conveniencia.

Por lo general, limitamos nuestro interés a los nodos internos de un árbol rojo-negro, ya que contienen los valores clave. En el resto de este capítulo, omitiremos las hojas cuando dibujemos árboles rojo-negros, como se muestra en la figura 13.1(c).

Llamamos al número de nodos negros en cualquier camino simple desde, pero sin incluir, un nodo x hasta una hoja, la altura negra del nodo, denotada  $bh.x$ . Por la propiedad 5, la noción de altura del negro está bien definida, ya que todos los caminos simples descendentes desde el nodo tienen el mismo número de nodos negros. Definimos la altura negra de un árbol rojo-negro como la altura negra de su raíz.

El siguiente lema muestra por qué los árboles rojo-negro son buenos árboles de búsqueda.

#### Lema 13.1

Un árbol rojo-negro con n nodos internos tiene una altura máxima de  $2 \lg n C 1$ .

Prueba Comenzamos mostrando que el subárbol arraigado en cualquier nodo x contiene al menos  $2bh.x / 1$  nodos internos. Probamos esta afirmación por inducción sobre la altura de x. Si la altura de x es 0, entonces x debe ser una hoja (T:nil), y el subárbol con raíz en x contiene al menos  $2bh.x / 1 D 20$  1 D 0 nodos internos. Para el paso inductivo, considere un nodo x que tiene altura positiva y es un nodo interno con dos hijos.

Cada niño tiene una altura de negro de  $bh.x / 0 bh.x / 1$ , dependiendo de si su color es rojo o negro, respectivamente. Dado que la altura de un hijo de x es menor que la altura de x misma, podemos aplicar la hipótesis inductiva para concluir que cada hijo tiene al menos  $2bh.x / 1 / 1$  nodos internos. Por lo tanto, el subárbol con raíz en x contiene al menos  $.2bh.x / 1 / 1 / C.2bh.x / 1 / 1 / C1 D 2bh.x / 1$  nodos internos, lo que prueba la afirmación.

Para completar la prueba del lema, sea h la altura del árbol. De acuerdo con la propiedad 4, al menos la mitad de los nodos en cualquier camino simple desde la raíz a una hoja, no

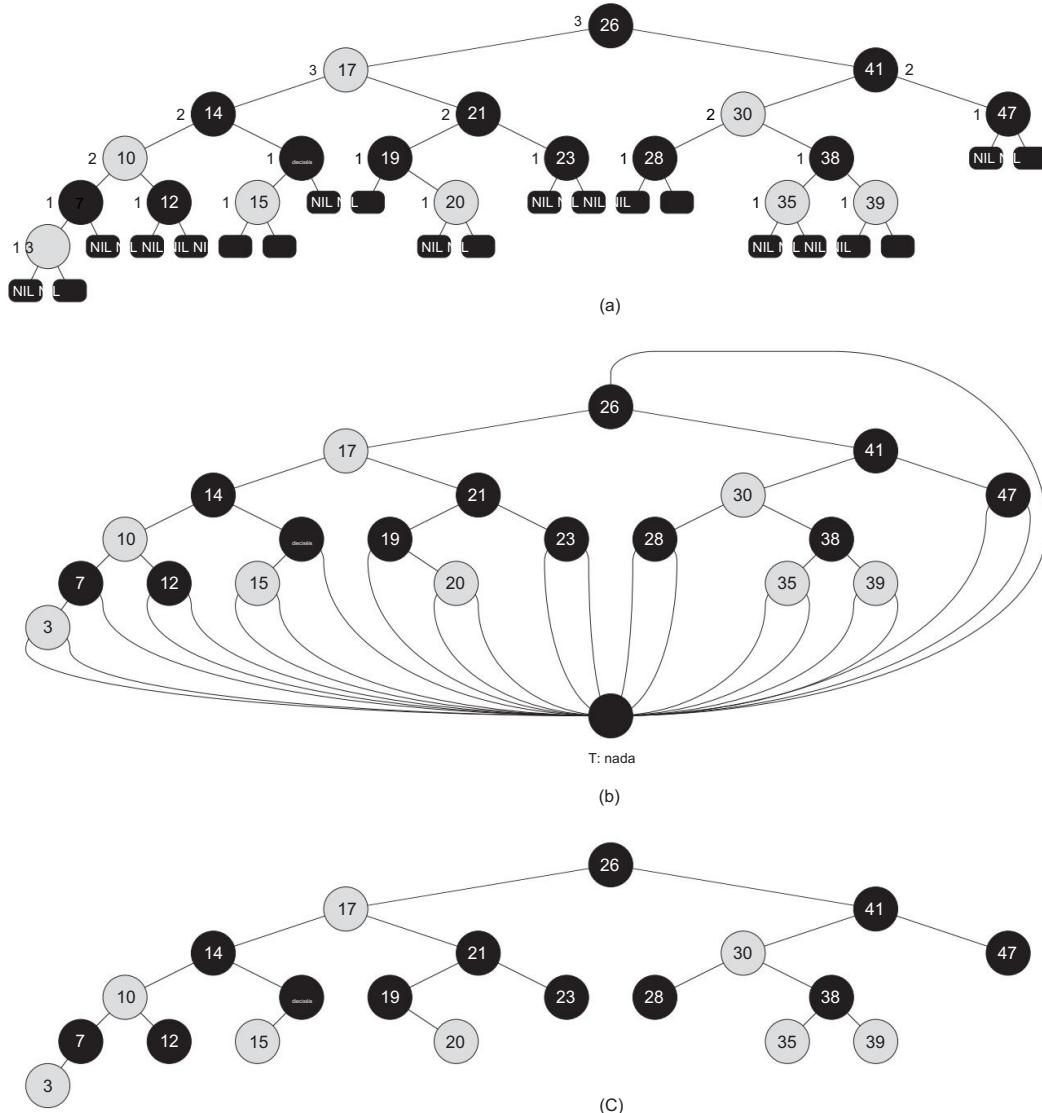


Figura 13.1 Un árbol rojo-negro con nodos negros oscurecidos y nodos rojos sombreados. Cada nodo en un árbol rojo-negro es rojo o negro, los hijos de un nodo rojo son ambos negros, y cada camino simple de un nodo a una hoja descendiente contiene el mismo número de nodos negros. (a) Cada hoja, que se muestra como NIL, es negra. Cada nodo que no es NIL está marcado con su altura negra; Los NIL tienen una altura de negro 0. (b) El mismo árbol rojo-negro pero con cada NIL reemplazado por el único centinela T:nil, que siempre es negro, y con las alturas de negro omitidas. El padre de la raíz también es el centinela. (c) El mismo árbol rojo-negro pero con hojas y el padre de la raíz omitido por completo. Usaremos este estilo de dibujo en el resto de este capítulo.

incluida la raíz, debe ser de color negro. En consecuencia, la altura del negro de la raíz debe ser al menos  $h=2$ ; de este modo,

$$n \geq 2^h - 1$$

Moviendo el 1 al lado izquierdo y tomando logaritmos en ambos lados se obtiene  $\lg n \leq h \leq \lg n + 1$ . ■

Como consecuencia inmediata de este lema, podemos implementar las operaciones de conjunto dinámico BUSCAR, MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR en tiempo  $O(\lg n)$  en árboles rojo-negro, ya que cada uno puede ejecutarse en tiempo  $O(h)$  en un binario árbol de búsqueda de altura  $h$  (como se muestra en el Capítulo 12) y cualquier árbol rojo-negro en  $n$  nodos es un árbol de búsqueda binario con altura  $O(\lg n)$ . (Por supuesto, las referencias a NIL en los algoritmos del Capítulo 12 tendrían que ser reemplazadas por T:nil.) Aunque los algoritmos TREE-INSERT y TREE-DELETE del Capítulo 12 se ejecutan en tiempo  $O(\lg n)$  cuando se les da un rojo -árbol negro como entrada, no admiten directamente las operaciones de conjunto dinámico INSERTAR y ELIMINAR, ya que no garantizan que el árbol de búsqueda binario modificado sea un árbol rojo-negro. Veremos en las Secciones 13.3 y 13.4, sin embargo, cómo soportar estas dos operaciones en  $O(\lg n)$  time.

### Ejercicios

#### 13.1-1

Al estilo de la figura 13.1(a), dibuje el árbol de búsqueda binario completo de altura 3 en las teclas f1; 2; : ; 15 g. Agregue las hojas NIL y coloree los nodos de tres maneras diferentes, de modo que las alturas negras de los árboles rojo-negro resultantes sean 2, 3 y 4.

#### 13.1-2

Dibuje el árbol rojo-negro que resulta después de llamar a TREE-INSERT en el árbol en Figura 13.1 con clave 36. Si el nodo insertado está coloreado de rojo, ¿el árbol resultante es un árbol rojo-negro? ¿Y si es de color negro?

#### 13.1-3

Definamos un árbol rojo-negro relajado como un árbol binario de búsqueda que satisface las propiedades rojo-negro 1, 3, 4 y 5. En otras palabras, la raíz puede ser roja o negra. Considere un árbol relajado rojo-negro T cuya raíz es roja. Si coloreamos la raíz de T de negro pero no hacemos otros cambios en T , ¿El árbol resultante es un árbol rojo-negro?

#### 13.1-4

Supongamos que “absorbemos” cada nodo rojo de un árbol rojo-negro en su padre negro, de modo que los hijos del nodo rojo se conviertan en hijos del padre negro. (Ignore lo que sucede con las teclas). ¿Cuáles son los posibles grados de un nodo negro después de todo?

sus niños rojos son absorbidos? ¿Qué puedes decir sobre la profundidad de las hojas del árbol resultante?

#### 13.1-5

Demuestre que el camino simple más largo desde un nodo  $x$  en un árbol rojo-negro hasta una hoja descendiente tiene una longitud como máximo el doble que el camino simple más corto desde el nodo  $x$  hasta una hoja descendiente.

#### 13.1-6

¿Cuál es el mayor número posible de nodos internos en un árbol rojo-negro con altura negra  $k$ ? ¿Cuál es el número más pequeño posible?

#### 13.1-7

Describa un árbol rojo-negro en  $n$  claves que tenga la mayor proporción posible de nodos internos rojos a nodos internos negros. ¿Cuál es esta proporción? ¿Qué árbol tiene la proporción más pequeña posible y cuál es la proporción?

---

## 13.2 Rotaciones

Las operaciones de árbol de búsqueda TREE-INSERT y TREE-DELETE, cuando se ejecutan en un árbol rojo negro con  $n$  claves, toman  $O.\lg n/$  tiempo. Debido a que modifican el árbol, el resultado puede violar las propiedades rojo-negro enumeradas en la Sección 13.1. Para restaurar estas propiedades, debemos cambiar los colores de algunos de los nodos del árbol y también cambiar la estructura del puntero.

Cambiamos la estructura del puntero a través de la rotación, que es una operación local en un árbol de búsqueda que conserva la propiedad del árbol de búsqueda binaria. La figura 13.2 muestra los dos tipos de rotaciones: rotaciones a la izquierda y rotaciones a la derecha. Cuando hacemos una rotación a la izquierda en un nodo  $x$ , asumimos que su hijo derecho y no es T:nil;  $x$  puede ser cualquier nodo del árbol cuyo hijo derecho no sea T:nil. La rotación a la izquierda "pivota" alrededor del vínculo de  $x$  a  $y$ . Hace que  $y$  sea la nueva raíz del subárbol, con  $x$  como el hijo izquierdo de  $y$  y el hijo izquierdo de  $y$  como el hijo derecho de  $x$ .

El pseudocódigo para LEFT-ROTATE asume que  $x:\text{right} \neq \text{T:nil}$  y que el padre de la raíz es T:nil.

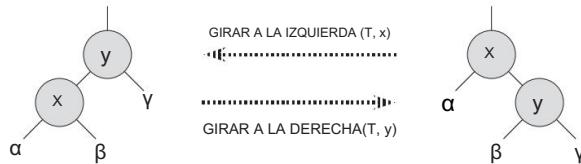


Figura 13.2 Las operaciones de rotación en un árbol de búsqueda binaria. La operación LEFT-ROTATE.T;  $x/y$  transforma la configuración de los dos nodos de la derecha en la configuración de la izquierda cambiando un número constante de punteros. La operación inversa RIGHT-ROTATE.T;  $y/x$  transforma la configuración de la izquierda en la configuración de la derecha. Las letras  $\alpha, \beta, \gamma$  representan subárboles arbitrarios. Una operación de rotación conserva la propiedad del árbol de búsqueda binaria: las claves en  $\alpha$  preceden a  $x:\text{key}$ , que precede a las claves en  $\beta$ , que preceden a  $y:\text{key}$ , que precede a las claves en  $\gamma$ .

GIRAR A LA IZQUIERDA.T;

```

x/ 1 y D x:derecha 2           // establecer
x:derecha D y:izquierda 3 si   y // convertir el subárbol izquierdo de y en el subárbol derecho de x
y:izquierda = T:nil 4
y:izquierda:p D x 5 y:p D x:p
6 si x:p == T: nil 7           // vincular el padre de x con y
T:root D y 8 elseif x ==      T:root D y
x:p:left x:p:left D y 9 10 else
x:p:right D y 11 y:left D x 12 x:p
D y

// pon x a la izquierda de y

```

La Figura 13.3 muestra un ejemplo de cómo LEFT-ROTATE modifica un árbol de búsqueda binaria. El código para RIGHT-ROTATE es simétrico. Tanto LEFT-ROTATE como RIGHT ROTATE se ejecutan en tiempo O(1). Solo los punteros se cambian por una rotación; todos los demás atributos en un nodo siguen siendo los mismos.

### Ejercicios

#### 13.2-1

Escriba el pseudocódigo para RIGHT-ROTATE.

#### 13.2-2

Argumente que en cada árbol de búsqueda binario de  $n$  nodos, hay exactamente  $n - 1$  rotaciones posibles.

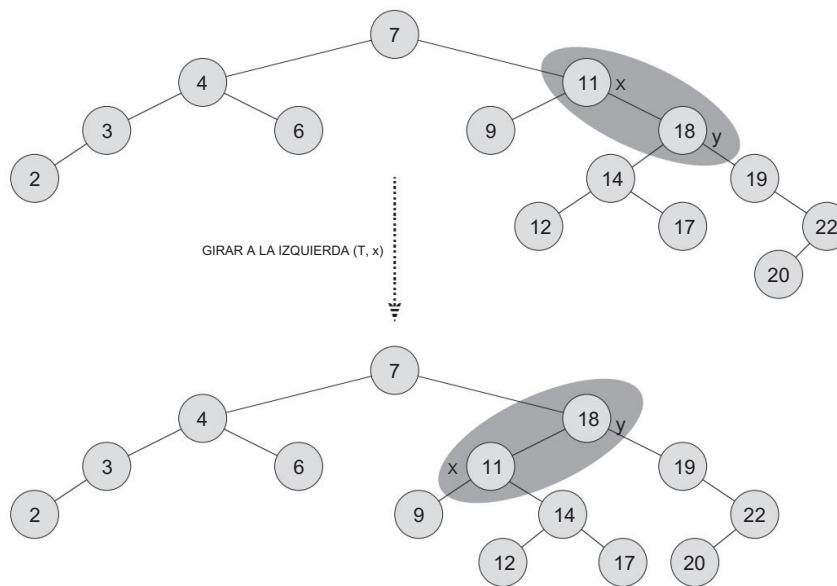


Figura 13.3 Un ejemplo de cómo funciona el procedimiento LEFT-ROTATE.T; x/ modifica un árbol de búsqueda binario. Los recorridos de árbol en orden del árbol de entrada y el árbol modificado producen la misma lista de valores clave.

#### 13.2-3

Sean a, b y c nodos arbitrarios en los subárboles „, ^ y , respectivamente, en el árbol izquierdo de la figura 13.2. ¿Cómo cambian las profundidades de a, b y c cuando se realiza una rotación a la izquierda en el nodo x de la figura?

#### 13.2-4

Muestre que cualquier árbol de búsqueda binario arbitrario de n nodos puede transformarse en cualquier otro árbol de búsqueda binario arbitrario de n nodos usando rotaciones On/. (Sugerencia: primero demuestre que como máximo n 1 rotaciones a la derecha son suficientes para transformar el árbol en una cadena que va a la derecha).

#### 13.2-5 ?

Decimos que un árbol de búsqueda binario T1 se puede convertir a la derecha en un árbol de búsqueda binario T2 si es posible obtener T2 de T1 a través de una serie de llamadas a RIGHT-ROTATE. Dé un ejemplo de dos árboles T1 y T2 tales que T1 no se pueda convertir por la derecha en T2.

Luego, muestre que si un árbol T1 se puede convertir a la derecha en T2, se puede convertir a la derecha usando llamadas O.n2/ a RIGHT-ROTATE.

### 13.3 Inserción

Podemos insertar un nodo en un árbol rojo-negro de  $n$  nodos en  $O.\lg n$  tiempo. Para hacerlo, usamos una versión ligeramente modificada del procedimiento TREE-INSERT (Sección 12.3) para insertar el nodo  $\tau$  en el árbol  $T$  como si fuera un árbol de búsqueda binaria ordinario, y luego coloreamos  $\tau$  rojo. (El ejercicio 13.3-1 le pide que explique por qué elegimos hacer que el nodo sea rojo en lugar de negro). Para garantizar que se conserven las propiedades rojo-negro, llamamos a un procedimiento auxiliar RB-INSERT-FIXUP para volver a colorear los nodos y realizar rotaciones . . La llamada RB-INSERT.T;  $\tau$  inserta el nodo  $\tau$ , cuya clave se supone que ya ha sido completada, en el árbol rojo-negro  $T$

```
RB-INSERTAR.T;  $\tau$ 
  1 y D T:nil 2 x
  D T:root 3 while
    x  $\neq$  T:nil 4 y D x if  $\tau$ :key
     $\leq$  x:key x D x:left
    5           else x D x:right 8
    6            $\tau$ :p D y 9 if y
    7           == T:nil 10 T:root D
     $\tau$  11 elseif
     $\tau$ :key  $<$  y:key 12
    y:left D  $\tau$  13 else y:right
    D  $\tau$  14  $\tau$ :left D T:nil 15  $\tau$ :right
    D T :nil 16  $\tau$ :color D
    ROJO 17 RB-INSERT-
    FIXUP.T;  $\tau$ 
```

Los procedimientos TREE-INSERT y RB-INSERT difieren en cuatro aspectos. Primero, todas las instancias de NIL en TREE-INSERT se reemplazan por T:nil. En segundo lugar, establecemos  $\tau$ :left y  $\tau$ :right en T:nil en las líneas 14–15 de RB-INSERT, para mantener la estructura de árbol adecuada. Tercero, coloreamos  $\tau$  rojo en la línea 16. Cuarto, debido a que colorear  $\tau$  rojo puede causar una violación de una de las propiedades rojo-negro, llamamos RB-INSERT-FIXUP.T;  $\tau$  en la línea 17 de RB-INSERT para restaurar el puntal rojo-negro erties.

```

RB-INSERTAR-ARREGLAR.T; '/

1 while ':p:color == RED 2 if ':p ==
':p:p:left 3 y D ':p:p:right 4 if y:color
== RED 5 ':p:color D BLACK 6 y:color
D NEGRO 7 ':p:p:color D ROJO 8 ' D
':p:p else if ' == ':p:derecha           // caso 1 //
caso 1 //                                caso 1 //
caso 1 //                                caso 1
caso 1

9
10          ' D ':p                         // caso 2 //
11          GIRO A LA IZQUIERDA.T;          caso 2 //
12          '/ ':p:color D NEGRO            caso 3 //
13          ':p:p:color D ROJO GIRAR       caso 3 //
14          A LA DERECHA.T; páginas/        caso 3
15          else (igual que la cláusula " then
           " con "derecha" e "izquierda" intercambiadas)
16 T:raíz:color D NEGRO

```

Para comprender cómo funciona RB-INSERT-FIXUP , dividiremos nuestro examen del código en tres pasos principales. Primero, determinaremos qué violaciones de las propiedades rojo-negro se introducen en RB-INSERT cuando el nodo ' se inserta y se colorea de rojo. En segundo lugar, examinaremos el objetivo general del ciclo while en las líneas 1 a 15. Finalmente, exploraremos cada uno de los tres casos<sup>1</sup> dentro del cuerpo del bucle while y veremos cómo logran el objetivo. La figura 13.4 muestra cómo opera RB INSERT-FIXUP en un árbol rojo-negro de muestra.

¿Cuál de las propiedades rojo-negro podría violarse al llamar a RB INSERT-FIXUP? La propiedad 1 ciertamente sigue siendo válida, al igual que la propiedad 3, ya que ambos hijos del nodo rojo recién insertado son el centinela T:nil. La propiedad 5, que dice que el número de nodos negros es el mismo en cada camino simple desde un nodo dado, también se cumple, porque el nodo ' reemplaza al centinela (negro) y el nodo ' es rojo con los hijos centinela. Por lo tanto, las únicas propiedades que podrían violarse son la propiedad 2, que requiere que la raíz sea negra, y la propiedad 4, que dice que un nodo rojo no puede tener un hijo rojo. Ambas posibles infracciones se deben a que ' está coloreada en rojo. La propiedad 2 se viola si ' es la raíz, y la propiedad 4 se viola si el padre de ' es rojo. La figura 13.4(a) muestra una violación de la propiedad 4 después de insertar el nodo '.

---

<sup>1</sup>El caso 2 cae dentro del caso 3, por lo que estos dos casos no son mutuamente excluyentes.

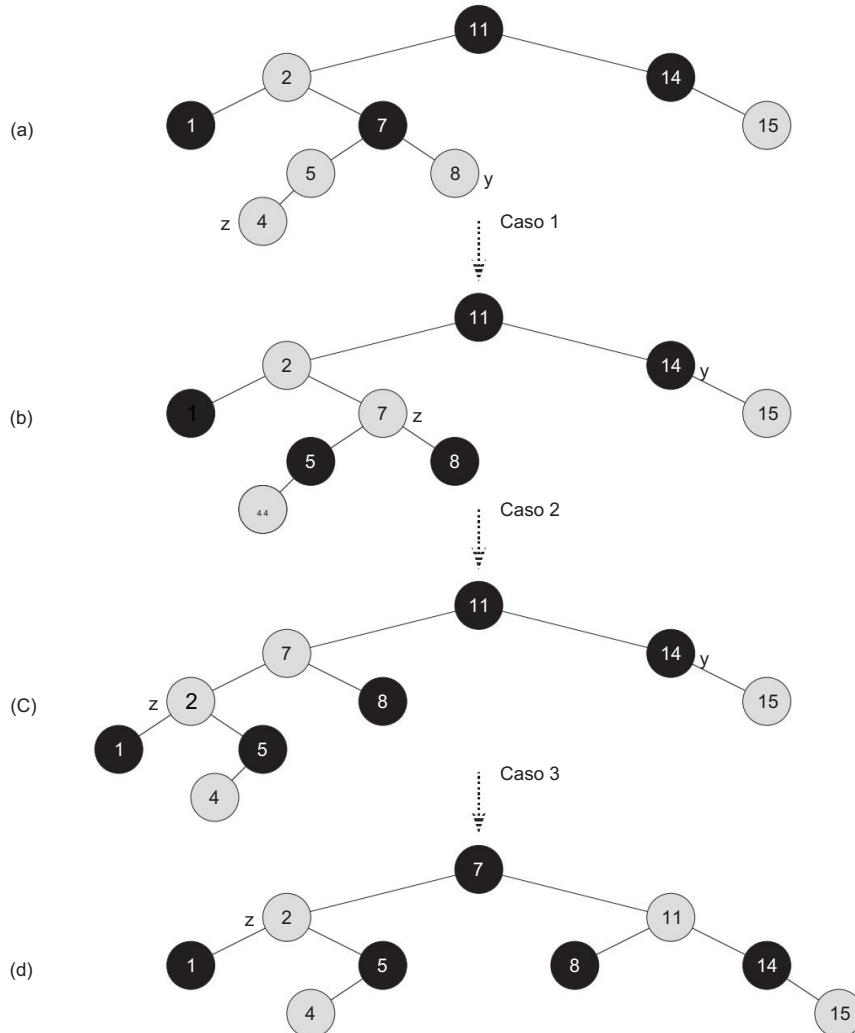


Figura 13.4 El funcionamiento de RB-INSERT-FIXUP. (a) Un nodo  $\zeta$  después de la inserción. Debido a que tanto  $\zeta$  como su padre  $\zeta:p$  son rojos, se produce una violación de la propiedad 4. Dado que el tío y de  $\zeta$  es rojo, se aplica el caso 1 en el código. Recoloreamos los nodos y movemos el puntero  $\zeta$  hacia arriba en el árbol, lo que da como resultado el árbol que se muestra en (b). Una vez más,  $\zeta$  y su padre son rojos, pero el tío y de  $\zeta$  es negro. Dado que  $\zeta$  es el hijo derecho de  $\zeta:p$ , se aplica el caso 2. Realizamos una rotación a la izquierda y el árbol resultante se muestra en (c). Ahora,  $\zeta$  es el hijo izquierdo de su padre, y se aplica el caso 3. El cambio de color y la rotación a la derecha producen el árbol en (d), que es un árbol legal rojo-negro.

El bucle while de las líneas 1 a 15 mantiene el siguiente invariante de tres partes en el inicio de cada iteración del ciclo:

- a. El nodo  $\cdot$  es rojo.
- b. Si  $\cdot:p$  es la raíz, entonces  $\cdot:p$  es negro.
- C. Si el árbol viola alguna de las propiedades rojo-negro, viola como máximo una de ellas, y la violación es de la propiedad 2 o de la propiedad 4. Si el árbol viola la propiedad 2, es porque  $\cdot$  es la raíz y es rojo. Si el árbol viola la propiedad 4, es porque tanto  $\cdot$  como  $\cdot:p$  son rojos.

La parte (c), que se ocupa de las violaciones de las propiedades rojo-negro, es más central para mostrar que RB-INSERT-FIXUP restaura las propiedades rojo-negro que las partes (a) y (b), que usamos en el camino para comprender situaciones en el código. Debido a que nos enfocaremos en el nodo  $\cdot$  y los nodos cercanos a él en el árbol, ayuda saber de la parte (a) que  $\cdot$  es rojo. Usaremos la parte (b) para mostrar que el nodo  $\cdot:p:p$  existe cuando lo referenciamos en las líneas 2, 3, 7, 8, 13 y 14.

Recuerde que necesitamos mostrar que un ciclo invariante es verdadero antes de la primera iteración del ciclo, que cada iteración mantiene el ciclo invariante y que el ciclo invariante nos da una propiedad útil al final del ciclo.

Comenzamos con los argumentos de inicialización y terminación. Luego, mientras examinamos con más detalle cómo funciona el cuerpo del ciclo, argumentaremos que el ciclo mantiene el invariante en cada iteración. En el camino, también demostraremos que cada iteración del ciclo tiene dos posibles resultados: o el puntero  $\cdot$  se mueve hacia arriba en el árbol, o realizamos algunas rotaciones y luego el ciclo termina.

**Inicialización:** antes de la primera iteración del ciclo, comenzamos con un árbol rojo-negro sin violaciones y agregamos un nodo rojo  $\cdot$ . Mostramos que cada parte del invariante se mantiene en el momento en que se llama RB-INSERT-FIXUP :

- a. Cuando se llama a RB-INSERT-FIXUP,  $\cdot$  es el nodo rojo que se agregó. b. Si  $\cdot:p$  es la raíz, entonces  $\cdot:p$  comenzó en negro y no cambió antes de la llamada de RB-INSERT-FIXUP.
- C. Ya hemos visto que las propiedades 1, 3 y 5 se cumplen cuando se llama a RB-INSERT FIXUP .

Si el árbol viola la propiedad 2, entonces la raíz roja debe ser el nodo recién agregado  $\cdot$ , que es el único nodo interno del árbol. Debido a que el padre y ambos hijos de  $\cdot$  son el centinela, que es negro, el árbol tampoco viola la propiedad 4. Por lo tanto, esta violación de la propiedad 2 es la única violación de las propiedades rojo-negro en todo el árbol.

Si el árbol viola la propiedad 4, entonces, debido a que los hijos del nodo  $\cdot$  son centinelas negros y el árbol no tuvo otras violaciones antes de agregar  $\cdot$ , el

la violación debe deberse a que tanto '`:p`' como '`:p:p`' están en rojo. Además, el árbol no viola otras propiedades rojo-negras.

**Terminación:** Cuando el bucle termina, lo hace porque '`:p`' es negro. (Si '`:p`' es la raíz, entonces '`:p`' es el centinela `T:nil`, que es negro). Por lo tanto, el árbol no viola la propiedad 4 en la terminación del bucle. Por el bucle invariante, la única propiedad que podría fallar es la propiedad 2. La línea 16 también restaura esta propiedad, de modo que cuando termina RB-INSERT-FIXUP, todas las propiedades rojo-negro se mantienen.

**Mantenimiento:** En realidad necesitamos considerar seis casos en el bucle while, pero tres de ellos son simétricos a los otros tres, dependiendo de si la línea 2 determina que el padre de '`:p`' sea un hijo izquierdo o un hijo derecho de '`:p`' el abuelo de '`:p:p`'.

Hemos dado el código solo para la situación en la que '`:p`' es un hijo izquierdo. El nodo '`:p:p`' existe, ya que por la parte (b) del bucle invariante, si '`:p`' es la raíz, entonces '`:p`' es negro.

Como entramos en una iteración de bucle solo si '`:p`' es rojo, sabemos que '`:p`' no puede ser la raíz. Por lo tanto, '`:p:p`' existe.

Distinguimos el caso 1 de los casos 2 y 3 por el color del hermano o "tío" de los padres de '`:p`'. La línea 3 hace que y apunte al tío de '`:p:p:right`', y la línea 4 prueba el color de `y`. Si `y` es rojo, entonces ejecutamos el caso 1. De lo contrario, el control pasa a los casos 2 y 3. En los tres casos, el abuelo '`:p:p`' es negro, ya que su padre '`:p`' es rojo y la propiedad 4 se viola sólo entre '`:y`' y '`:p`'.

#### Caso 1: el tío y de '`:p`' es rojo

figura 13.5 muestra la situación del caso 1 (líneas 5 a 8), que ocurre cuando tanto '`:p`' como `y` son rojos. Debido a que '`:p:p`' es negro, podemos colorear tanto '`:p`' como `y` de negro, solucionando así el problema de que '`:y`' y '`:p`' sean rojos, y podemos colorear '`:p:p`' de rojo, manteniendo así la propiedad 5. Luego repetimos el ciclo while con '`:p:p`' como el nuevo nodo '`:p`'. El puntero '`:p`' sube dos niveles en el árbol.

Ahora mostramos que el caso 1 mantiene el bucle invariante al comienzo de la siguiente iteración. Usamos '`:p`' para denotar nodo '`:p`' en la iteración actual, y '`:p'0`' para denotar el nodo que se llamará nodo '`:p`' en la prueba en la línea 1 en la próxima iteración.

- Debido a que esta iteración colorea '`:p:p`' rojo, el nodo '`:p'0`' es rojo al comienzo del siguiente iteración.
- El nodo '`:p'0`' es '`:p:p:p`' en esta iteración, y el color de este nodo no cambia. Si este nodo es la raíz, era negro antes de esta iteración y permanece negro al comienzo de la próxima iteración.
- Ya hemos argumentado que el caso 1 mantiene la propiedad 5, y no introducir una violación de las propiedades 1 o 3.

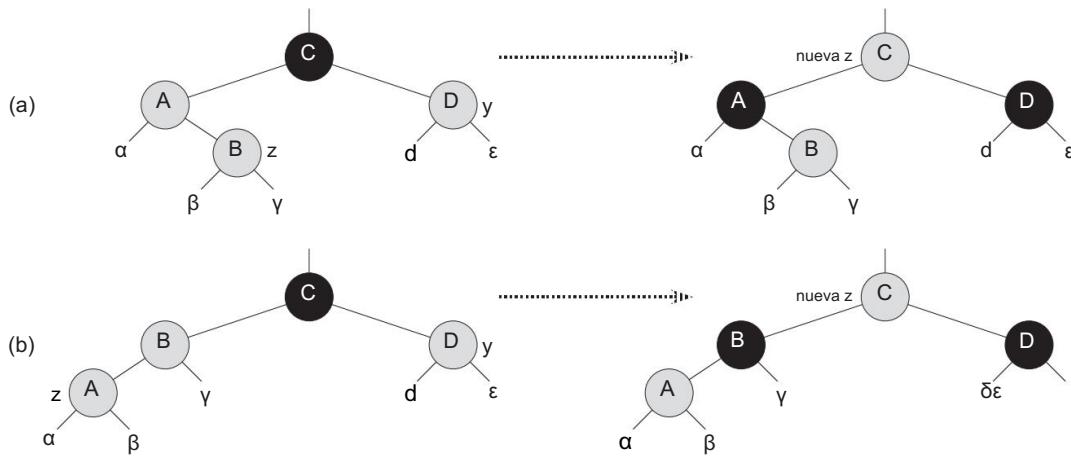


Figura 13.5 Caso 1 del procedimiento RB-INSERT-FIXUP. Se viola la propiedad 4, ya que 'y' y su padre ':p' son rojos. Tomamos la misma acción si (a) 'y' es un hijo derecho o (b) 'y' es un hijo izquierdo. Cada uno de los subárboles .., , , , y " tiene una raíz negra, y cada uno tiene la misma altura negra.

El código del caso 1 cambia los colores de algunos nodos, conservando la propiedad 5: todos los caminos simples descendentes desde un nodo hasta una hoja tienen el mismo número de negros. El ciclo while continúa con el abuelo del nodo ':p:p como el nuevo '. Cualquier violación de la propiedad 4 ahora puede ocurrir solo entre el nuevo ', que es rojo, y su padre, si también es rojo.

Si el nodo '0 es la raíz al comienzo de la siguiente iteración, entonces el caso 1 corrigió la única violación de la propiedad 4 en esta iteración. Como '0 es rojo y es la raíz, la propiedad 2 se convierte en la única que se viola, y esta violación se debe a '0 .

Si el nodo '0 no es la raíz al comienzo de la siguiente iteración, entonces el caso 1 no ha creado una violación de la propiedad 2. El caso 1 corrigió la única violación de la propiedad 4 que existía al comienzo de esta iteración. Entonces hizo '0 rojo y dejó '0 :p solo. Si '0 :p fuera negro, no hay violación de la propiedad 4. Si '0 :p era rojo, colorear '0 rojo creaba una violación de la propiedad 4 entre '0 y '0 :p.

Caso 2: El tío y de ' es negro y ' es hijo derecho Caso

3: El tío y de ' es negro y ' es hijo izquierdo En los

casos 2 y 3, el color del tío y es negro. Distinguimos los dos casos según que ' sea hijo derecho o izquierdo de ':p. Las líneas 10 y 11 constituyen el caso 2, que se muestra en la figura 13.6 junto con el caso 3. En el caso 2, el nodo ' es un hijo derecho de su padre. Inmediatamente usamos una rotación a la izquierda para transformar la situación en el caso 3 (líneas 12 a 14), en el que el nodo ' es un hijo izquierdo. Porque

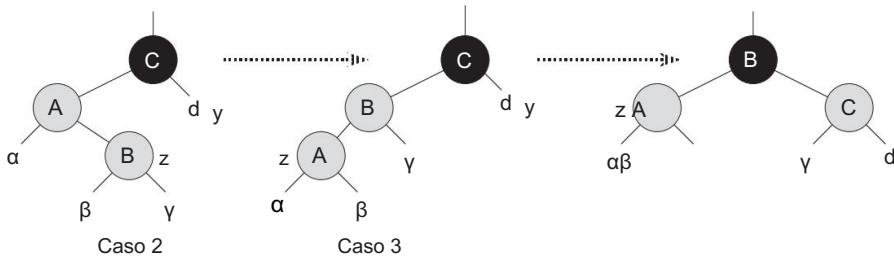


Figura 13.6 Casos 2 y 3 del procedimiento RB-INSERT-FIXUP. Como en el caso 1, la propiedad 4 se viola en el caso 2 o en el caso 3 porque  $\text{\'e}'$  y su padre  $\text{'p}$  son rojos. Cada uno de los subárboles  $\text{\'e}'$ ,  $\text{\'e}''$  e  $\text{\'e}'''$  tiene una raíz negra ( $\text{\'e}', \text{\'e}''$  y de la propiedad 4, y  $\text{\'e}'''$  porque de lo contrario estaríamos en el caso 1), y cada uno tiene la misma altura negra. Transformamos el caso 2 en el caso 3 mediante una rotación a la izquierda, que conserva la propiedad 5: todos los caminos simples descendentes desde un nodo hasta una hoja tienen el mismo número de negros. El caso 3 provoca algunos cambios de color y una rotación a la derecha, que también conservan la propiedad 5. El ciclo while termina, porque se cumple la propiedad 4: ya no hay dos nodos rojos seguidos.

tanto '`:p`' como '`:p`' son rojos, la rotación no afecta ni a la altura del negro de los nodos ni a la propiedad 5. Ya sea que ingresemos el caso 3 directamente o a través del caso 2, el tío y de '`:p`' es negro, ya que de lo contrario habríamos ejecutado el caso 1 Además, existe el nodo '`:p:p`', ya que hemos argumentado que este nodo existía en el momento en que se ejecutaron las líneas 2 y 3, y después de subir '`:p`' un nivel en la línea 10 y luego bajar un nivel en la línea 11, la identidad de '`:p:p`' permanece sin cambios. En el caso 3, ejecutamos algunos cambios de color y una rotación a la derecha, que preservan la propiedad 5, y luego, como ya no tenemos dos nodos rojos seguidos, terminamos. El ciclo `while` no itera otra vez, ya que '`:p`' ahora es negro.

Ahora mostramos que los casos 2 y 3 mantienen el bucle invariante. (Como acabamos de argumentar, ':p será negro en la próxima prueba en la línea 1, y el cuerpo del ciclo no se ejecutará nuevamente).

- a. El caso 2 hace que ':p' apunte a ':p', que es rojo. En los casos 2 y 3 no se producen más cambios en ':p' o en su color.
  - b. El caso 3 hace que ':p' sea negra, de modo que si ':p' es la raíz al comienzo de la siguiente iteración, es negro. C.

Como en el caso 1, las propiedades 1, 3 y 5 se mantienen en los casos 2 y 3.

Dado que el nodo  $\gamma$  no es la raíz en los casos 2 y 3, sabemos que no hay violación de la propiedad 2. Los casos 2 y 3 no introducen una violación de la propiedad 2, ya que el único nodo que se vuelve rojo se convierte en un hijo de un nodo negro por la rotación en el caso 3.

Los casos 2 y 3 corregir la única violación de la propiedad 4 y no introducen otra violación.

Habiendo demostrado que cada iteración del bucle mantiene el invariante, hemos demostrado que RB-INSERT-FIXUP restaura correctamente las propiedades rojo-negro.

## Análisis

¿ Cuál es el tiempo de ejecución de RB-INSERT? Dado que la altura de un árbol rojo-negro en  $n$  nodos es  $O.\lg n$ , las líneas 1–16 de RB-INSERT toman  $O.\lg n$ . En RB-INSERT FIXUP, el ciclo while se repite solo si ocurre el caso 1, y luego el puntero  $\text{p}$  se mueve dos niveles hacia arriba en el árbol. Por lo tanto , el número total de veces que se puede ejecutar el bucle while es  $O.\lg n$ . Por lo tanto, RB-INSERT toma un total de  $O.\lg n$  tiempo. Además, nunca realiza más de dos rotaciones, ya que el ciclo while termina si se ejecuta el caso 2 o el caso 3.

## Ejercicios

13.3-1

En la línea 16 de RB-INSERT, establecemos el color del nuevo nodo ' en rojo.

Observe que si hubiéramos elegido establecer el color de ' en negro, entonces no se violaría la propiedad 4 de un árbol rojo-negro. ¿Por qué no elegimos establecer el color de ' en negro?

13.3-2

Muestre los árboles rojo-negros que resultan después de insertar sucesivamente las llaves 41; 38; 31; 12; 19; 8 en un árbol rojo-negro inicialmente vacío.

13.3-3

13.3-4

Al profesor Teach le preocupa que RB-INSERT-FIXUP pueda establecer  $T:\text{nil}:\text{color}$  en RED, en cuyo caso la prueba en la línea 1 no provocaría que el ciclo terminara cuando  $\tau$  es la raíz. Demuestre que la preocupación del profesor no tiene fundamento argumentando que RB INSERT-FIXUP nunca establece  $T:\text{nil}:\text{color}$  en RED.

13 3=5

Consideré un árbol rojo-negro formado al insertar  $n$  nodos con RB-INSERT. Argumente que si  $n > 1$ , el árbol tiene al menos un nodo rojo.

13 3-6

Sugiera cómo implementar RB-INSERT de manera eficiente si la representación de árboles rojos y negros no incluye almacenamiento para números principales.

## 13.4 Eliminación

Al igual que las otras operaciones básicas en un árbol rojo-negro de  $n$  nodos, la eliminación de un nodo lleva tiempo  $O.\lg n$ . Eliminar un nodo de un árbol rojo-negro es un poco más complicado que insertar un nodo.

El procedimiento para borrar un nodo de un árbol rojo-negro se basa en el procedimiento BORRAR ÁRBOL (Sección 12.3). Primero, necesitamos personalizar la subrutina TRANSPLANT que llama TREE-DELETE para que se aplique a un árbol rojo-negro:

```
RB-TRASPLANTE.T; tu; / 1 si
u:p == T:nil 2 T:raíz
D 3 elseif u ==
u:p:izquierda 4 u:p:izquierda
D 5 else u:p:derecha
D 6 :p D u:p
```

El procedimiento RB-TRANSPLANT se diferencia del TRASPLANTE en dos aspectos. Primero, la línea 1 hace referencia al centinela `T:nil` en lugar de `NIL`. En segundo lugar, la asignación a `:p` en la línea 6 ocurre incondicionalmente: podemos asignar a `:p` incluso si apunta al centinela. De hecho, aprovecharemos la capacidad de asignar a `:p` cuando `D T:nil`.

El procedimiento RB-DELETE es como el procedimiento TREE-DELETE, pero con líneas adicionales de pseudocódigo. Algunas de las líneas adicionales realizan un seguimiento de un nodo y que podría causar violaciones de las propiedades rojo-negro. Cuando queremos eliminar el nodo `y` que tiene menos de dos hijos, entonces `y` se elimina del árbol y queremos que `y` sea `x`. Cuando `x` tiene dos hijos, entonces `y` debería ser el sucesor de `x`, y `y` se mueve a la posición de `x` en el árbol. También recordamos el color de `y` antes de que se elimine o se mueva dentro del árbol, y hacemos un seguimiento del nodo `x` que se mueve a la posición original de `y` en el árbol, porque el nodo `x` también podría causar violaciones de las propiedades rojo-negro. Despues de eliminar el nodo `y`, RB-DELETE llama a un procedimiento auxiliar RB-DELETE-FIXUP, que cambia de color y realiza rotaciones para restaurar las propiedades rojo-negro.

```

RB-BORRAR.T; '/ 1 y D
  ' 2 y-color-
  original D y:color 3 if ':izquierda == T:nil
  4 x D ':derecha 5 RB-
  TRANSPLANT.T; ';
  ':derecha/ 6 elseif ':derecha == T:nil 7 x D ':izquierda
  8 RB-TRANSPLANT.T; ', ':left/ 9
  else y D TREE-MINIMUM.
  ':right/ y-original-color D y:color x D y:right if y:p == '
10
11
12
13      x:p D y más
14      RB-TRANSPLANTE.T; y; y:derecha/ y:derecha D
15      ':derecha y:derecha:p D
16      y
17      RB-TRASPLANTE.T; ', y/ y:izquierda
D ':izquierda 18 19
y:izquierda:p D y 20 y:color D
':color 21 if y-original-color == BLACK
22 RB-DELETE-FIXUP.T; X/

```

Aunque RB-DELETE contiene casi el doble de líneas de pseudocódigo que TREE-DELETE, los dos procedimientos tienen la misma estructura básica. Puede encontrar cada línea de TREE-DELETE dentro de RB-DELETE (con los cambios de reemplazar NIL por T:nil y reemplazar las llamadas a TRANSPLANT por llamadas a RB-TRANSPLANT), ejecutadas en las mismas condiciones.

Estas son las otras diferencias entre los dos procedimientos:

Mantenemos el nodo y como el nodo eliminado del árbol o movido dentro del árbol. La línea 1 hace que y apunte al nodo ' cuando ' tiene menos de dos hijos y, por lo tanto, se elimina. Cuando ' tiene dos hijos, la línea 9 hace que y apunte al sucesor de ', tal como en TREE-DELETE, y y se moverá a la posición de ' en el árbol.

Debido a que el color del nodo y puede cambiar, la variable y-original-color almacena el color de y antes de que ocurra cualquier cambio. Las líneas 2 y 10 establecen esta variable inmediatamente después de las asignaciones a y. Cuando ' tiene dos hijos, entonces y ≠ ' y el nodo y se mueve a la posición original del nodo ' en el árbol rojo-negro; la línea 20 da a y el mismo color que '. Necesitamos guardar el color original de y para probarlo en el

fin de RB-DELETE; si era negro, quitar o mover y podría causar violaciones de las propiedades rojo-negro.

Como se discutió, hacemos un seguimiento del nodo  $x$  que se mueve a la posición original del nodo  $y$ . Las asignaciones en las líneas 4, 7 y 11 hacen que  $x$  apunte al único hijo de  $y$  o, si  $y$  no tiene hijos, al centinela  $T:\text{nil}$ . (Recuerde de la Sección 12.3 que  $y$  no tiene hijo izquierdo).

Dado que el nodo  $x$  se mueve a la posición original del nodo  $y$ , el atributo  $x:p$  siempre se establece para señalar la posición original en el árbol del padre de  $y$ , incluso si  $x$  es, de hecho, el centinela  $T:\text{nil}$ . A menos que  $'$  sea el padre original de  $y$  (lo que ocurre solo cuando  $'$  tiene dos hijos y su sucesor  $y$  es el hijo derecho de  $'$ ), la asignación a  $x:p$  tiene lugar en la línea 6 de RB-TRANSPLANT. (Observe que cuando se llama a RB-TRANSPLANT en las líneas 5, 8 o 14, el segundo parámetro que se pasa es el mismo que  $x$ ).

Sin embargo, cuando el padre original de  $y$  es  $'$ , no queremos que  $x:p$  apunte al padre original de  $y$ , ya que estamos eliminando ese nodo del árbol. Debido a que el nodo  $y$  se moverá hacia arriba para tomar la posición de  $'$  en el árbol, establecer  $x:p$  en  $y$  en la línea 13 hace que  $x:p$  apunte a la posición original del padre de  $y$ , incluso si  $x D T:\text{nil}$ .

Finalmente, si el nodo  $y$  fuera negro, podríamos haber introducido una o más violaciones de las propiedades rojo-negro, por lo que llamamos a RB-DELETE-FIXUP en la línea 22 para restaurar las propiedades rojo-negro. Si  $y$  era rojo, las propiedades rojo-negro aún se mantienen cuando se elimina o mueve  $y$ , por las siguientes razones:

1. No ha cambiado ninguna altura de negro en el árbol.
2. No se han hecho adyacentes nodos rojos. Debido a que  $y$  toma el lugar de  $'$  en el árbol, junto con el color de  $'$ , no podemos tener dos nodos rojos adyacentes en la nueva posición de  $y$  en el árbol. Además, si  $y$  no era el hijo derecho de  $'$ , entonces el hijo derecho original  $x$  de  $y$  reemplaza a  $y$  en el árbol. Si  $y$  es rojo, entonces  $x$  debe ser negro, por lo que reemplazar  $y$  por  $x$  no puede causar que dos nodos rojos se vuelvan adyacentes.
3. Como  $y$  no podría haber sido la raíz si fuera roja, la raíz permanece negra.

Si el nodo  $y$  estuviera en negro, pueden surgir tres problemas, que la llamada de RB-DELETE FIXUP solucionará. Primero, si  $y$  había sido la raíz y un hijo rojo de  $y$  se convierte en la nueva raíz, hemos violado la propiedad 2. Segundo, si tanto  $x$  como  $x:p$  son rojos, entonces hemos violado la propiedad 4. Tercero, mover  $y$  dentro de la tree hace que cualquier camino simple que previamente contenía  $y$  tenga un nodo negro menos. Por lo tanto, la propiedad 5 ahora es violada por cualquier ancestro de  $y$  en el árbol. Podemos corregir la violación de la propiedad 5 diciendo que el nodo  $x$ , que ahora ocupa la posición original de  $y$ , tiene un negro "extra". Es decir, si sumamos 1 al recuento de nodos negros en cualquier ruta simple que contenga  $x$ , entonces, según esta interpretación, se cumple la propiedad 5. Cuando eliminamos o movemos el nodo negro  $y$ , "empujamos" su negrura hacia el nodo  $x$ . El problema es que ahora el nodo  $x$  no es ni rojo ni negro, violando así la propiedad 1. En cambio,

el nodo x es "dblemente negro" o "rojo y negro", y contribuye con 2 o 1, respectivamente, al recuento de nodos negros en caminos simples que contienen x. El atributo de color de x seguirá siendo ROJO (si x es rojo y negro) o NEGRO (si x es dblemente negro). En otras palabras, el negro adicional en un nodo se refleja en las x que apuntan al nodo en lugar de en el atributo de color .

Ahora podemos ver el procedimiento RB-DELETE-FIXUP y examinar cómo restaura las propiedades rojo-negro en el árbol de búsqueda.

#### RB-BORRAR-ARREGLAR.T;

```

x/ 1 while x ≠ T:root and x:color == BLACK 2 if x ==
x:p:left 3 w D x:p:right 4 if
w:color == RED 5 w:color D
BLACK 6 x :p:color D ROJO 7
GIRAR A LA IZQUIERDA.T; x:p/ 8 w D // caso
x:p:derecha 9 si w:izquierda:color ==
NEGRO y w:derecha:color == NEGRO 10 1 // caso
1 // caso
1 // caso 1

w:color D ROJO // caso
11 x D x:p 2 // caso 2
12 else if w:right:color == BLACK
13 w:left:color D BLACK // caso
14 w:color D RED 3 // caso
15 RIGHT-ROTATE.T; w/ w
    D x:p:right w:color 3 // caso
    16 D x:p:color x:p:color D 3 // caso
17 BLACK w:right:color D 4 // caso
18 BLACK 19 20 LEFT- 4 // caso
    ROTATE.T; x:p/ 21 x D T:root 22 else (igual
que la cláusula luego con "derecha" 4 // caso
e "izquierda" intercambiadas) 23 x:color D NEGRO 4 // caso 4

```

El procedimiento RB-DELETE-FIXUP restaura las propiedades 1, 2 y 4. Los ejercicios 13.4-1 y 13.4-2 le piden que muestre que el procedimiento restaura las propiedades 2 y 4, por lo que en el resto de esta sección nos centraremos en propiedad 1. El objetivo del ciclo while en las líneas 1 a 22 es mover el negro adicional hacia arriba en el árbol hasta que 1. x

apunte a un nodo rojo y negro, en cuyo caso coloreamos x (individualmente) negro en línea 23;

2. x apunta a la raíz, en cuyo caso simplemente "quitamos" el negro extra; o 3. después de haber realizado las rotaciones y cambios de color adecuados, salimos del bucle.

Dentro del bucle while , x siempre apunta a un nodo doblemente negro que no es raíz. Determinamos en la línea 2 si x es un hijo izquierdo o un hijo derecho de su padre x:p. (Hemos dado el código para la situación en la que x es un hijo izquierdo; la situación en la que x es un hijo derecho, la línea 22, es simétrica.) Mantenemos un puntero w al hermano de x. Dado que el nodo x es doblemente negro, el nodo w no puede ser T:nil, porque de lo contrario, el número de negros en el camino simple desde x:p hasta la hoja w (una sola negra) sería menor que el número en el camino simple desde x :p a x.

Los cuatro casos2 del código aparecen en la figura 13.7. Antes de examinar cada caso en detalle, veamos de forma más general cómo podemos verificar que la transformación en cada uno de los casos conserva la propiedad 5. La idea clave es que en cada caso, la transformación aplicada conserva el número de nodos negros (incluidos los nodos extra de x). negro) desde (e incluyendo) la raíz del subárbol que se muestra a cada uno de los subárboles ; ; ; . Por lo tanto, si la propiedad 5 se cumple antes de la transformación, continúa valiéndose después. Por ejemplo, en la figura 13.7(a), que ilustra el caso 1, el número de nodos negros desde la raíz hasta el subárbol , o ^ es 3, tanto antes como después de la transformación. (Nuevamente, recuerde que el nodo x agrega un negro adicional). De manera similar, el número de nodos negros desde la raíz hasta cualquiera de , l, ", y es 2, tanto antes como después de la transformación. En la figura 13.7(b), el conteo debe involucrar el valor c del atributo de color de la raíz del subárbol mostrado, que puede ser ROJO o NEGRO. Si definimos count.RED/ D 0 y count.BLACK/ D 1, entonces el número de nodos negros desde la raíz hasta , es 2 C count.c/, tanto antes como después de la transformación. En este caso, después de la transformación, el nuevo nodo x tiene el atributo de color c, pero este nodo es realmente rojo y negro (si c D ROJO) o doblemente negro (si c D NEGRO) . Puede verificar los otros casos de manera similar (ve

Caso 1: el hermano w de x es

rojo El caso 1 (líneas 5 a 8 de RB-DELETE-FIXUP y la figura 13.7(a)) ocurre cuando el nodo  $w$ , el hermano del nodo  $x$ , es rojo. Dado que  $w$  debe tener hijos negros, podemos cambiar los colores de  $w$  y  $x:p$  y luego realizar una rotación a la izquierda en  $x:p$  sin violar ninguna de las propiedades rojo-negro. El nuevo hermano de  $x$ , que es uno de los hijos de  $w$  antes de la rotación, ahora es negro y, por lo tanto, hemos convertido el caso 1 en el caso 2, 3 o 4.

Los casos 2, 3 y 4 ocurren cuando el nodo w es negro; se distinguen por los colores de los hijos de w.

2Al igual que en RB-INSERT-FIXUP, los casos en RB-DELETE-FIXUP no son mutuamente excluyentes.

Caso 2: el hermano w de x es negro, y ambos hijos de w son negros En el caso 2 (líneas 10-11 de RB-DELETE-FIXUP y Figura 13.7(b)), ambos hijos de w son negros. Dado que w también es negro, quitamos un negro de x y w, dejando x con solo un negro y dejando w rojo. Para compensar la eliminación de un negro de x y w, nos gustaría agregar un negro extra a x:p, que originalmente era rojo o negro. Lo hacemos repitiendo el ciclo while con x:p como el nuevo nodo x. Observe que si ingresamos del caso 2 al caso 1, el nuevo nodo x es rojo y negro, ya que el x:p original era rojo. Por lo tanto, el valor c del atributo de color del nuevo nodo x es ROJO, y el ciclo termina cuando prueba la condición del ciclo. Luego coloreamos el nuevo nodo x (individualmente) de negro en la línea 23.

Caso 3: el hermano w de x es negro, el hijo izquierdo de w es rojo y el hijo derecho de w es negro El caso 3 (líneas 13-16 y figura 13.7(c)) ocurre cuando w es negro, su hijo izquierdo es rojo y su hijo derecho es negro. Podemos cambiar los colores de w y su hijo izquierdo w:left y luego realizar una rotación a la derecha en w sin violar ninguna de las propiedades rojo-negro. El nuevo hermano w de x ahora es un nodo negro con un hijo derecho rojo y, por lo tanto, hemos transformado el caso 3 en el caso 4.

Caso 4: el hermano w de x es negro y el hijo derecho de w es rojo El caso 4 (líneas 17 a 21 y figura 13.7(d)) ocurre cuando el hermano w del nodo x es negro y el hijo derecho de w es rojo. Al hacer algunos cambios de color y realizar una rotación a la izquierda en x:p, podemos eliminar el negro adicional en x, haciéndolo negro por separado, sin violar ninguna de las propiedades rojo-negro. Establecer x para que sea la raíz hace que el ciclo while termine cuando prueba la condición del ciclo.

### Análisis

¿ Cuál es el tiempo de ejecución de RB-DELETE? Dado que la altura de un árbol rojo-negro de n nodos es  $O.\lg n$ , el costo total del procedimiento sin la llamada a RB-DELETE FIXUP toma  $O.\lg n / \text{tiempo}$ . Dentro de RB-DELETE-FIXUP, cada uno de los casos 1, 3 y 4 conducen a la terminación después de realizar un número constante de cambios de color y un máximo de tres rotaciones. El caso 2 es el único caso en el que se puede repetir el ciclo while , y luego el puntero x se mueve hacia arriba en el árbol como máximo  $O.\lg n / \text{veces}$ , sin realizar rotaciones. Por lo tanto, el procedimiento RB-DELETE-FIXUP toma  $O.\lg n / \text{tiempo}$  y realiza como máximo tres rotaciones, y el tiempo total para RB-DELETE es por lo tanto también  $O.\lg n / \text{.}$

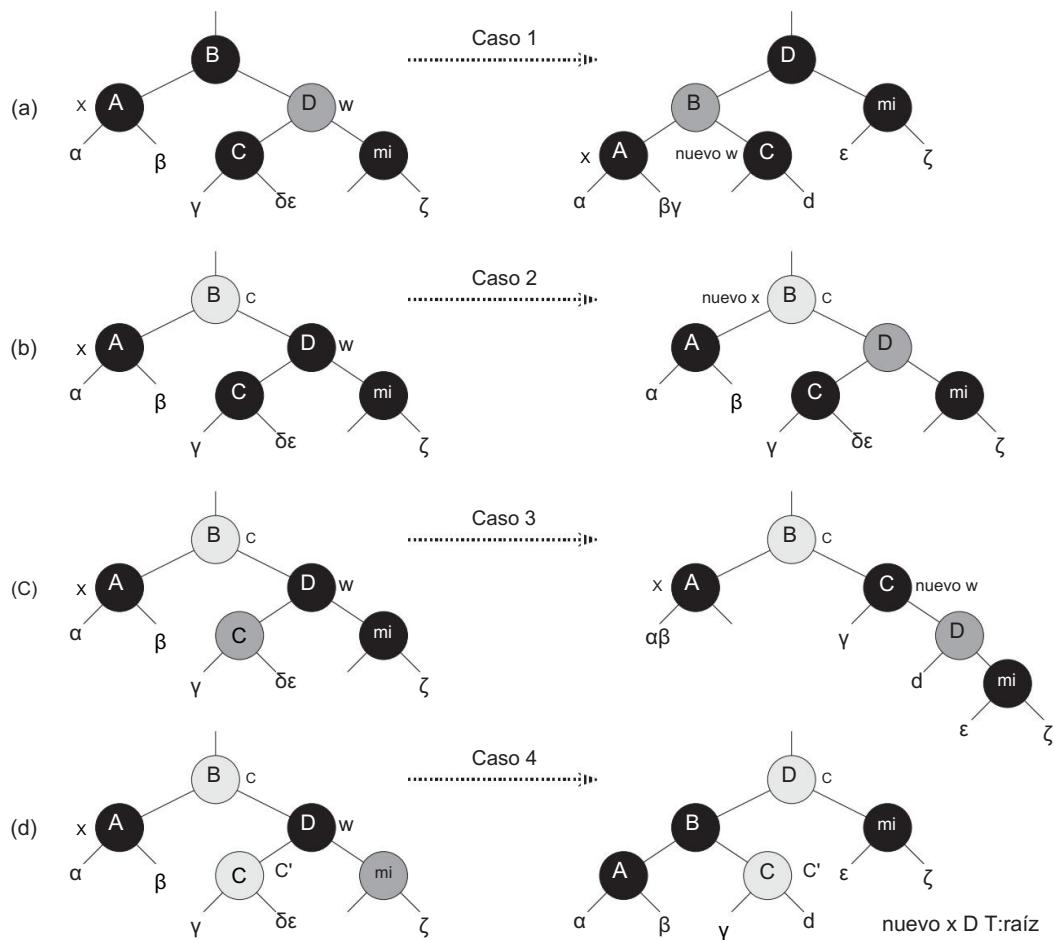


Figura 13.7 Los casos en el bucle while del procedimiento RB-DELETE-FIXUP. Los nodos oscurecidos tienen atributos de color NEGRO, los nodos muy sombreados tienen atributos de color ROJO y los nodos ligeramente sombreados tienen atributos de color representados por c y c0 que pueden ser ROJO o NEGRO. Las cartas, ; : : ; representan subárboles arbitrarios. Cada caso transforma la configuración de la izquierda en la configuración de la derecha cambiando algunos colores y/o realizando una rotación. Cualquier nodo señalado por x tiene un negro adicional y es doblemente negro o rojo y negro. Solo el caso 2 hace que el bucle se repita. (a) El caso 1 se transforma en el caso 2, 3 o 4 intercambiando los colores de los nodos B y D y realizando una rotación a la izquierda. (b) En el caso 2, el negro extra representado por el puntero x se mueve hacia arriba en el árbol coloreando el nodo D rojo y configurando x para que apunte al nodo B. Si ingresamos del caso 2 al caso 1, el bucle while termina porque el nuevo nodo x es rojo y negro y, por lo tanto, el valor c de su atributo de color es ROJO. (c) El caso 3 se transforma en el caso 4 intercambiando los colores de los nodos C y D y realizando una rotación a la derecha. (d) El caso 4 elimina el negro extra representado por x cambiando algunos colores y realizando una rotación a la izquierda (sin violar las propiedades rojo-negro), y luego el ciclo termina.

### Ejercicios

#### 13.4-1

Argumenta que después de ejecutar RB-DELETE-FIXUP, la raíz del árbol debe ser negra.

#### 13.4-2

Argumente que si en RB-DELETE tanto  $x$  como  $x:p$  son rojos, entonces la propiedad 4 se restaura mediante la llamada a RB-DELETE-FIXUP.T; X/.

#### 13.4-3

En el Ejercicio 13.3-2, encontró el árbol rojo-negro que resulta de insertar sucesivamente las claves 41; 38; 31; 12; 19; 8 en un árbol inicialmente vacío. Ahora muestre los árboles rojo-negro que resultan de la eliminación sucesiva de las claves en el orden 8; 12; 19; 31; 38; 41.

#### 13.4-4

¿ En qué líneas del código para RB-DELETE-FIXUP podríamos examinar o modificar el centinela T:nil?

#### 13.4-5

En cada uno de los casos de la figura 13.7, proporcione el recuento de nodos negros desde la raíz del subárbol que se muestra hasta cada uno de los subárboles ; ; ; ; ; y verifique que cada conteo permanezca igual después de la transformación. Cuando un nodo tiene un atributo de color c o c0

, use la notación cuenta.c/ o cuenta.c0 / simbólicamente en su conteo.

#### 13.4-6

A los profesores Skelton y Baron les preocupa que al comienzo del caso 1 de RB DELETE-FIXUP, el nodo  $x:p$  podría no ser negro. Si los profesores están en lo correcto, entonces las líneas 5 y 6 están equivocadas. Demuestre que  $x:p$  debe ser negro al comienzo del caso 1, para que los profesores no tengan nada de qué preocuparse.

#### 13.4-7

Suponga que se inserta un nodo  $x$  en un árbol rojo-negro con RB-INSERT y luego se elimina inmediatamente con RB-DELETE. ¿El árbol rojo-negro resultante es el mismo que el árbol rojo-negro inicial? Justifica tu respuesta.

---

## Problemas

### 13-1 Conjuntos dinámicos persistentes

Durante el curso de un algoritmo, a veces encontramos que necesitamos mantener versiones anteriores de un conjunto dinámico a medida que se actualiza. A tal conjunto lo llamamos persistente. Una forma de implementar un conjunto persistente es copiar todo el conjunto cada vez que se modifica, pero este enfoque puede ralentizar un programa y también consumir mucho espacio. A veces, podemos hacerlo mucho mejor.

Considere un conjunto persistente  $S$  con las operaciones INSERTAR, ELIMINAR y BUSCAR, que implementamos usando árboles de búsqueda binarios como se muestra en la figura 13.8(a). Mantenemos una raíz separada para cada versión del conjunto. Para insertar la clave 5 en el conjunto, creamos un nuevo nodo con la clave 5. Este nodo se convierte en el hijo izquierdo de un nuevo nodo con la clave 7, ya que no podemos modificar el nodo existente con la clave 7.

De manera similar, el nuevo nodo con clave 7 se convierte en el hijo izquierdo de un nuevo nodo con clave 8 cuyo hijo derecho es el nodo existente con clave 10. El nuevo nodo con clave 8 se convierte, a su vez, en el hijo derecho de una nueva raíz  $r_0$  con clave 4 cuyo hijo izquierdo es el nodo existente con la clave 3. Por lo tanto, copiamos solo una parte del árbol y compartimos algunos de los nodos con el árbol original, como se muestra en la figura 13.8(b).

Suponga que cada nodo del árbol tiene los atributos key, left y right , pero no padre.  
(Véase también el ejercicio 13.3-6.)

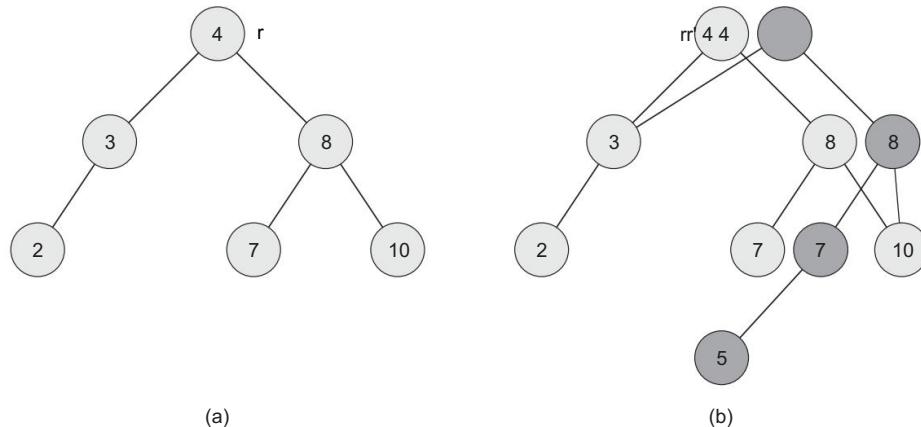


Figura 13.8 (a) Un árbol de búsqueda binaria con claves 2; 3; 4; 7; 8; 10. (b) El árbol de búsqueda binaria persistente que resulta de la inserción de la clave 5. La versión más reciente del conjunto consta de los nodos accesibles desde la raíz  $r_0$ . Los nodos sombreados se \_\_\_\_\_, y la versión anterior consta de los nodos accesibles desde  $r$ . Fuertemente agregan cuando se inserta la clave 5.

- a. Para un árbol de búsqueda binaria persistente general, identifique los nodos que necesitamos cambie para insertar una clave k o eliminar un nodo y.
- b. Escriba un procedimiento INSERTAR-ÁRBOL-PERSISTENTE que, dado un árbol persistente T que y una clave k para insertar, devuelve un nuevo árbol persistente  $T'$  sea el resultado de insertar  $T$  en  $k$  en  $T$ .
- C. Si la altura del árbol de búsqueda binaria persistente T es  $h$ , ¿cuáles son los requisitos de tiempo y espacio de su implementación de PERSISTENT-TREE-INSERT?  
(El requisito de espacio es proporcional al número de nuevos nodos asignados).
- d. Supongamos que hubiéramos incluido el atributo padre en cada nodo. En este caso, PERSISTENT-TREE-INSERT necesitaría realizar una copia adicional. Demuestre que PERSISTENT-TREE-INSERT requeriría entonces  $O(n)$  tiempo y espacio, donde  $n$  es el número de nodos en el árbol.
- mi. Muestre cómo usar árboles rojo-negro para garantizar que el tiempo de ejecución en el peor de los casos y el espacio son  $O(\lg n)$  por inserción o eliminación.

### 13-2 Operación de unión en árboles rojo-negro

La operación de unión toma dos conjuntos dinámicos S1 y S2 y un elemento x tal que para cualquier  $x \in S1$  y  $x \notin S2$ , tenemos  $x:clave \ x:clave$ . Devuelve un conjunto SD  $S1 \cup S2$ . En este problema, investigamos cómo implementar la operación de combinación en árboles rojo-negro.

- a. Dado un árbol rojo-negro  $T$ , almacenemos su altura de negro como el nuevo atributo  $T:nh$ . Argumente que RB-INSERT y RB-DELETE pueden mantener el atributo  $nh$  sin requerir almacenamiento adicional en los nodos del árbol y sin aumentar los tiempos de ejecución asintóticos. Muestre que mientras desciende por  $T$  termine la altura negra de cada nodo que visitamos en  $O(1)$  tiempo por nodo visitado.
- Deseamos implementar la operación RB-JOIN( $T1;T2$ ), que destruye  $T1$  y  $T2$  y devuelve un árbol rojo-negro  $TD = T1 \cup T2$ . Sea  $n$  el número total de nodos en  $T1$  y  $T2$ .
- b. Suponga que  $T1:nh = T2:nh$ . Describa un algoritmo  $O(\lg n)$ -time que encuentre un nodo negro y en  $T1$  con la clave más grande entre aquellos nodos cuya altura negra es  $T2:nh$ .
- C. Sea  $Ty$  el subárbol con raíz en  $y$ . Describa cómo  $Ty$  puede reemplazar a  $Ty$  en el tiempo  $O(1)$  sin destruir la propiedad del árbol de búsqueda binaria.
- d. ¿De qué color debemos hacer  $x$  para que se mantengan las propiedades rojo-negro 1, 3 y 5?  
Describa cómo hacer cumplir las propiedades 2 y 4 en  $O(\lg n)$  tiempo.

mi. Argumente que no se pierde generalidad al hacer la suposición de la parte (b). Describe la situación simétrica que surge cuando  $T_1: \text{bh } T_2: \text{bh}$ .

F. Argumente que el tiempo de ejecución de RB-JOIN es  $O.\lg n/.$

### 13-3 Árboles AVL Un

árbol AVL es un árbol de búsqueda binaria con altura equilibrada: para cada nodo  $x$ , las alturas de los subárboles izquierdo y derecho de  $x$  difieren como máximo en 1. Para implementar un árbol AVL, mantenemos un atributo adicional en cada nodo:  $x:h$  es la altura del nodo  $x$ . En cuanto a cualquier otro árbol de búsqueda binaria  $T$  , asumimos que  $T:\text{root}$  apunta al nodo raíz.

- Demuestre que un árbol AVL con  $n$  nodos tiene una altura  $O.\lg n/$ . (Sugerencia: demuestre que un árbol AVL de altura  $h$  tiene al menos nodos  $F_h$  , donde  $F_h$  es el  $h$ -ésimo número de Fibonacci).
- Para insertar en un árbol AVL, primero colocamos un nodo en el lugar apropiado en el orden del árbol de búsqueda binaria. Posteriormente, es posible que el árbol ya no esté equilibrado en altura. Específicamente, las alturas de los hijos izquierdo y derecho de algún nodo pueden diferir en 2. Describa un procedimiento  $\text{BALANCE}.x/$ , que toma un subárbol con raíz en  $x$  cuyos hijos izquierdo y derecho están equilibrados en altura y tienen alturas que difieren en como máximo 2 , es decir,  $jx:right:h x:left:hj 2$ , y altera el subárbol enraizado en  $x$  para equilibrarlo en altura. (Sugerencia: use rotaciones).

C. Usando la parte (b), describa un procedimiento recursivo  $\text{AVL-INSERT}.x; '/$  que toma un nodo  $x$  dentro de un árbol AVL y un nodo recién creado ' (cuya clave ya ha sido ingresada), y agrega ' al subárbol enraizado en  $x$ , manteniendo la propiedad de que  $x$  es la raíz de un árbol AVL . Como en  $\text{TREE-INSERT}$  de la Sección 12.3, suponga que ':key ya ha sido ingresada y que ':left D NIL y ':right D NIL; suponga también que ':h D 0. Por lo tanto, para insertar el nodo ' en el árbol AVL  $T$

, llamamos  $\text{AVL-INSERT}.T:\text{raíz}; '/$ .

- Muestre que  $\text{AVL-INSERT}$ , ejecutado en un árbol AVL de  $n$  nodos, toma un tiempo  $O.\lg n/$  y realiza rotaciones  $O.1/$ .

### 13-4 Treaps Si

insertamos un conjunto de  $n$  elementos en un árbol de búsqueda binaria, el árbol resultante puede estar terriblemente desequilibrado, lo que lleva a largos tiempos de búsqueda. Sin embargo, como vimos en la Sección 12.4, los árboles de búsqueda binarios construidos aleatoriamente tienden a estar balanceados. Por lo tanto, una estrategia que, en promedio, crea un árbol equilibrado para un conjunto fijo de elementos sería permutar aleatoriamente los elementos y luego insertarlos en ese orden en el árbol. ¿Qué pasa si no tenemos todos los artículos a la vez? Si recibimos los artículos de uno en uno tiempo, ¿podemos seguir construyendo al azar un árbol de búsqueda binaria a partir de ellos?

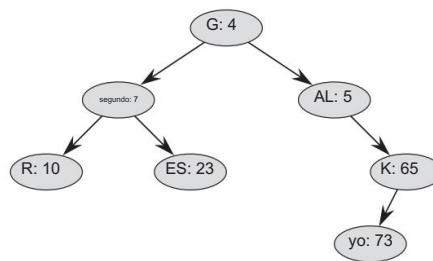


Figura 13.9 Una trampa. Cada nodo  $x$  está etiquetado con  $x:\text{clave} : x:\text{prioridad}$ . Por ejemplo, la raíz tiene clave G y prioridad 4.

Examinaremos una estructura de datos que responde afirmativamente a esta pregunta. Un treap es un árbol de búsqueda binario con una forma modificada de ordenar los nodos. La figura 13.9 muestra un ejemplo. Como es habitual, cada nodo  $x$  del árbol tiene un valor de clave  $x:\text{clave}$ . Además, asignamos  $x:\text{priority}$ , que es un número aleatorio elegido de forma independiente para cada nodo. Suponemos que todas las prioridades son distintas y también que todas las claves son distintas. Los nodos del treap están ordenados de modo que las claves obedezcan la propiedad del árbol de búsqueda binaria y las prioridades obedezcan la propiedad de orden del montón mínimo:

Si es un hijo izquierdo de  $u$ , entonces  $:key < u:key$ .

Si es hijo derecho de  $u$ , entonces  $:key > u:key$ .

Si es hijo de  $u$ , entonces  $:prioridad > u:prioridad$ .

(Esta combinación de propiedades es la razón por la cual el árbol se denomina "trampa": tiene características tanto de un árbol de búsqueda binaria como de un montón).

Es útil pensar en las trampas de la siguiente manera. Supongamos que insertamos nodos  $x_1; x_2; \dots; x_n$ , con claves asociadas, en un treap. Entonces, el treap resultante es el árbol que se habría formado si los nodos se hubieran insertado en un árbol de búsqueda binaria normal en el orden dado por sus prioridades (elegidas al azar), es decir,  $x_i : \text{priority} < x_j : \text{priority}$  significa que habíamos insertado  $x_i$  antes de  $x_j$ .

- Demostrar que dado un conjunto de nodos  $x_1; x_2; \dots; x_n$ , con claves y prioridades asociadas, todas distintas, el tratamiento asociado con estos nodos es único.
- Muestre que la altura esperada de un treap es  $\lg n$ , y por lo tanto el tiempo esperado para buscar un valor en el treap es  $\lg n$ .

Veamos cómo insertar un nuevo nodo en un treap existente. Lo primero que hacemos es asignar al nuevo nodo una prioridad aleatoria. Luego llamamos al algoritmo de inserción, al que llamamos TREAP-INSERT, cuyo funcionamiento se ilustra en la figura 13.10.

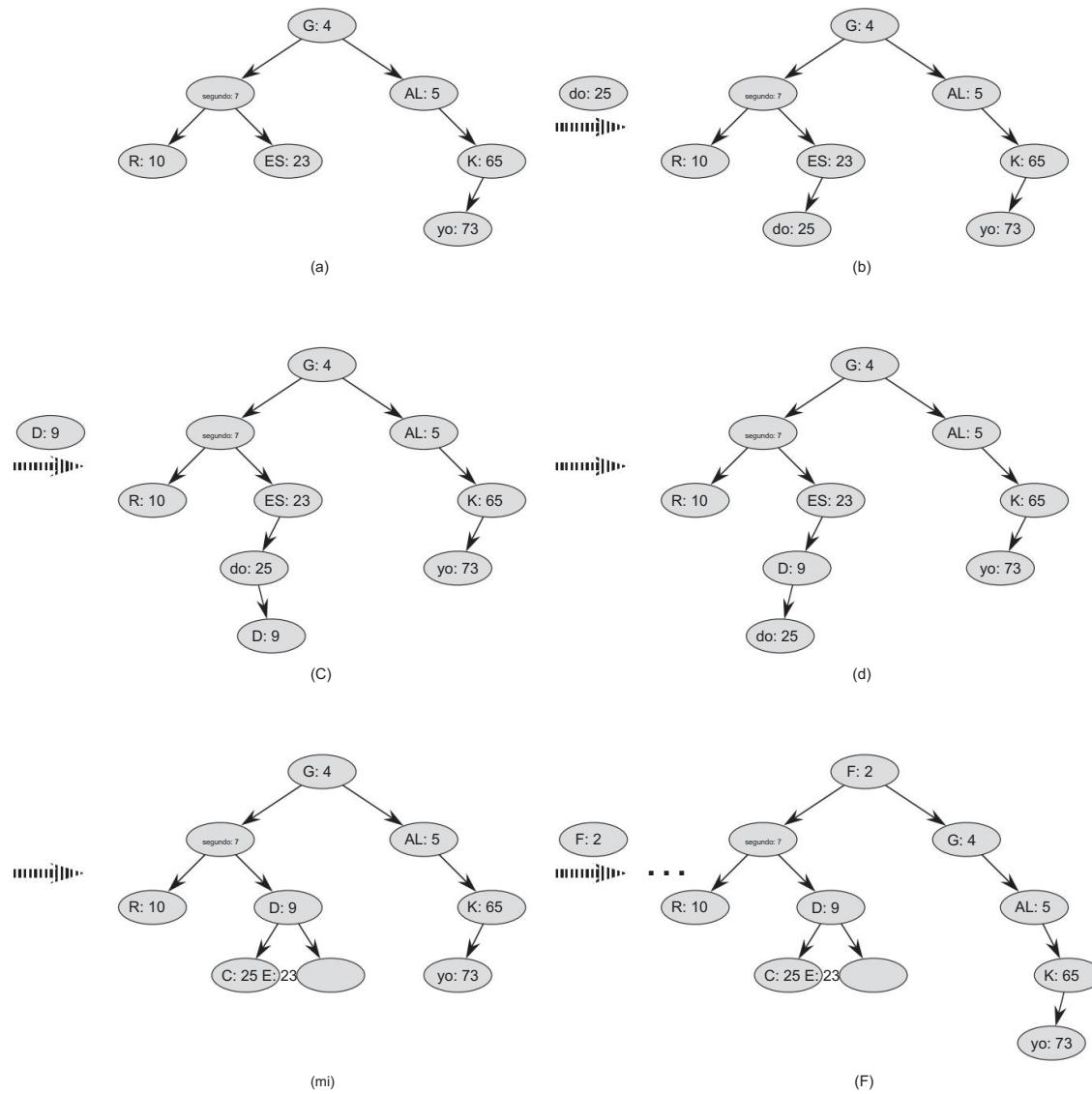


Figura 13.10 La operación de TREAP-INSERT. (a) El tratamiento original, antes de la inserción. (b) El treap después de insertar un nodo con clave C y prioridad 25. (c)–(d) Etapas intermedias al insertar un nodo con clave D y prioridad 9. (e) El treap después de la inserción de las partes (c) y (d) está hecho. (f) El treap después de insertar un nodo con clave F y prioridad 2.

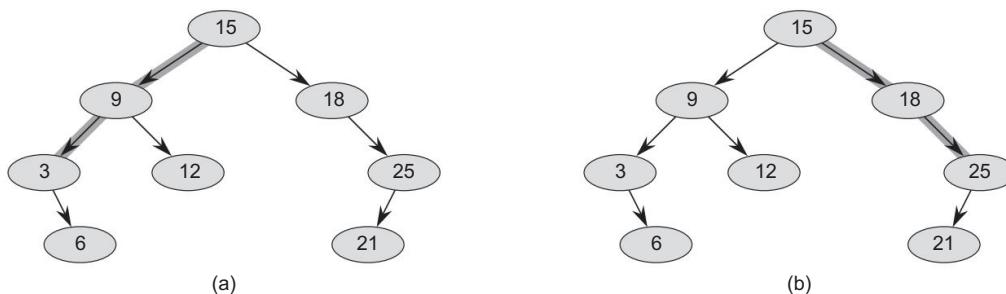


Figura 13.11 Espinas de un árbol de búsqueda binaria. El lomo izquierdo está sombreado en (a) y el lomo derecho está sombreado en (b).

C. Explique cómo funciona TREAP-INSERT . Explique la idea en inglés y dé pseu docode. (Sugerencia: ejecute el procedimiento habitual de inserción de árbol de búsqueda binaria y luego realice rotaciones para restaurar la propiedad de orden de montón mínimo).

d. Demuestre que el tiempo de ejecución esperado de TREAP-INSERT es  $\lg n$ .

TREAP-INSERT realiza una búsqueda y luego una secuencia de rotaciones. Aunque estas dos operaciones tienen el mismo tiempo de ejecución esperado, en la práctica tienen costos diferentes. Una búsqueda lee información del treap sin modificarla.

Por el contrario, una rotación cambia los punteros principales y secundarios dentro del treap. En la mayoría de las computadoras, las operaciones de lectura son mucho más rápidas que las operaciones de escritura. Por lo tanto, nos gustaría que TREAP-INSERT realice pocas rotaciones. Mostraremos que el número esperado de rotaciones realizadas está acotado por una constante.

Para hacerlo, necesitaremos algunas definiciones, que se muestran en la figura 13.11.

La columna izquierda de un árbol de búsqueda binario T es el camino simple desde la raíz hasta el nodo con la clave más pequeña. En otras palabras, la columna izquierda es el camino simple desde la raíz que consta solo de bordes izquierdos. Simétricamente, la columna derecha de T es el camino simple desde la raíz que consta solo de bordes derechos. La longitud de una columna vertebral es el número de nodos que contiene.

mi. Considere el treap T inmediatamente después de que TREAP-INSERT haya insertado el nodo x.

Sea C la longitud de la columna derecha del subárbol izquierdo de x. Sea D la longitud de la columna izquierda del subárbol derecho de x. Demuestre que el número total de rotaciones que se realizaron durante la inserción de x es igual a  $C + D$ .

Ahora calcularemos los valores esperados de C y D. Sin pérdida de generalidad, asumimos que las claves son  $1; 2; \dots; n$ , ya que los estamos comparando solo entre sí.

Para los nodos  $x$  e  $y$  en treap  $T$  donde  $y \neq x$ , sea  $k_D$   $x:\text{key}$  e  $i_D$   $y:\text{key}$ . Definimos variables aleatorias indicadoras

$X_{ik}$   $D$   $y$  está en el lomo derecho del subárbol izquierdo de  $x$ :

F. Muestre que  $X_{ik} D 1$  si y solo si  $y:\text{prioridad} > x:\text{prioridad}$ ,  $y:\text{clave} < x:\text{clave}$ ,  $y$ , para cada  $\ell$  tal que  $y:\text{clave} < \ell:\text{clave} < x:\text{clave}$ , tenemos  $y:\text{prioridad} < \text{prioridad}$ .

gramo. Muestra esa

$$\Pr fX_{ik} D 1 g D = \frac{\sum_{\substack{k1 \\ j1 \\ jD1}} \frac{1}{k1} \cdot \frac{1}{j1} \cdot \frac{1}{jD1}}{\sum_{\substack{k1 \\ j1 \\ jD1}} \frac{1}{k1}}$$

H. Muestra esa

$$E \Omega E D X_{jj} C 1 = \frac{1}{nk} \sum_{\substack{k1 \\ j1 \\ jD1}} \frac{1}{k1} \cdot \frac{1}{j1} \cdot \frac{1}{jD1}$$

i. Use un argumento de simetría para demostrar que

$$E \Omega E D 1 = \frac{1}{nk} \sum_{\substack{k1 \\ j1 \\ jD1}} \frac{1}{k1} \cdot \frac{1}{j1} \cdot \frac{1}{jD1}$$

j. Concluya que el número esperado de rotaciones realizadas al insertar un nodo en un treap es menor que 2.

## Notas del capítulo

La idea de balancear un árbol de búsqueda se debe a Adel'son-Vel'ski y Landis [2], quienes introdujeron una clase de árboles de búsqueda balanceados llamados "árboles AVL" en 1962, descritos en el Problema 13-3. Otra clase de árboles de búsqueda, denominada "árboles 2-3", fue introducida por JE Hopcroft (inédito) en 1970. Un árbol 2-3 mantiene el equilibrio mediante la manipulación de los grados de los nodos del árbol. El Capítulo 18 cubre una generalización de 2-3 árboles introducidos por Bayer y McCreight [35], llamados "árboles B".

Los árboles rojo-negro fueron inventados por Bayer [34] bajo el nombre de "árboles B binarios simétricos". Guibas y Sedgewick [155] estudiaron detalladamente sus propiedades e introdujeron la convención de color rojo/negro. Andersson [15] da un código más simple

variante de árboles rojo-negros. Weiss [351] llama a esta variante árboles AA. Un árbol AA es similar a un árbol rojo-negro excepto que los hijos izquierdos nunca pueden ser rojos.

Los treaps, el tema del problema 13-4, fueron propuestos por Seidel y Aragon [309].

Son la implementación predeterminada de un diccionario en LEDA [253], que es una colección bien implementada de estructuras de datos y algoritmos.

Hay muchas otras variaciones en los árboles binarios balanceados, incluidos los árboles de peso balanceado [264], los árboles k-vecinos [245] y los árboles de chivo expiatorio [127]. Tal vez los más intrigantes sean los "árboles abiertos" introducidos por Sleator y Tarjan [320], que son "autoajustables". (Ver Tarjan [330] para una buena descripción de los árboles abiertos.)

Los árboles separados mantienen el equilibrio sin ninguna condición de equilibrio explícita, como el color. En cambio, las "operaciones de visualización" (que involucran rotaciones) se realizan dentro del árbol cada vez que se realiza un acceso. El costo amortizado (vea el Capítulo 17) de cada operación en un árbol de  $n$  nodos es  $O.\lg n$ .

Las listas de omisión [286] proporcionan una alternativa a los árboles binarios equilibrados. Una lista de omisión es una lista vinculada que se aumenta con una serie de punteros adicionales. Cada operación de diccionario se ejecuta en el tiempo esperado  $O.\lg n$  en una lista de omisión de  $n$  elementos.

---

## 14

## Aumento de estructuras de datos

Algunas situaciones de ingeniería no requieren más que una estructura de datos de "libro de texto", como una lista doblemente enlazada, una tabla hash o un árbol de búsqueda binaria, pero muchas otras requieren una pizca de creatividad. Sin embargo, solo en raras situaciones necesitará crear un tipo de estructura de datos completamente nuevo. Más a menudo, bastará con aumentar la estructura de datos de un libro de texto almacenando información adicional en él. A continuación, puede programar nuevas operaciones para que la estructura de datos admita la aplicación deseada. Sin embargo, aumentar una estructura de datos no siempre es sencillo, ya que las operaciones ordinarias en la estructura de datos deben actualizar y mantener la información agregada.

Este capítulo analiza dos estructuras de datos que construimos al aumentar los árboles rojos y negros. La sección 14.1 describe una estructura de datos que admite operaciones estadísticas de orden general en un conjunto dinámico. Entonces podemos encontrar rápidamente el  $i$ -ésimo número más pequeño en un conjunto o el rango de un elemento dado en el ordenamiento total del conjunto. La Sección 14.2 resume el proceso de aumento de una estructura de datos y proporciona un teorema que puede simplificar el proceso de aumento de árboles rojo-negro. La sección 14.3 usa este teorema para ayudar a diseñar una estructura de datos para mantener un conjunto dinámico de intervalos, como los intervalos de tiempo. Dado un intervalo de consulta, podemos encontrar rápidamente un intervalo en el conjunto que lo superponga.

---

### 14.1 Estadísticas dinámicas de pedidos

El capítulo 9 introdujo la noción de una estadística de orden. En concreto, el estadístico de  $i$ -ésimo orden de un conjunto de  $n$  elementos, donde  $i \in \{1, 2, \dots, n\}$ , es simplemente el elemento del conjunto con la  $i$ -ésima clave más pequeña. Vimos cómo determinar cualquier estadística de orden en  $O(n)$  tiempo a partir de un conjunto desordenado. En esta sección, veremos cómo modificar árboles rojo-negro para que podamos determinar cualquier estadístico de orden para un conjunto dinámico en  $O(\lg n)$  tiempo. También veremos cómo calcular el rango de un elemento, su posición en el orden lineal del conjunto, en  $O(\lg n)$  tiempo.

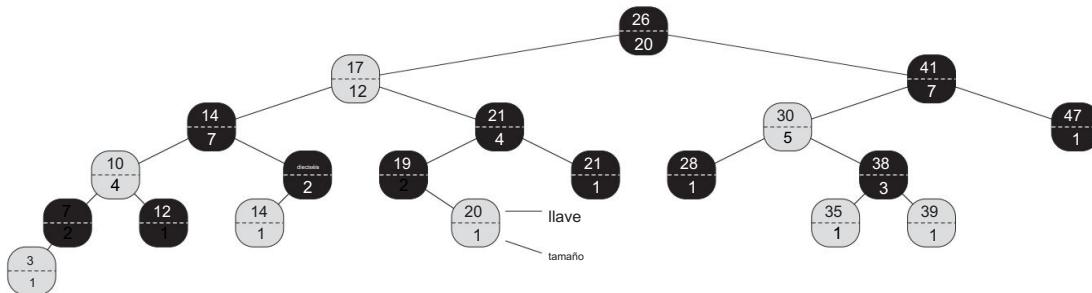


Figura 14.1 Un árbol estadístico de orden, que es un árbol rojo-negro aumentado. Los nodos sombreados son rojos y los nodos oscurecidos son negros. Además de sus atributos habituales, cada nodo  $x$  tiene un atributo  $x:\text{tamaño}$ , que es el número de nodos, además del centinela, en el subárbol con raíz en  $x$ .

La figura 14.1 muestra una estructura de datos que puede admitir operaciones rápidas de estadísticas de pedidos. Un árbol de estadísticas de orden T es simplemente un árbol rojo-negro con información adicional almacenada en cada nodo. Además de los atributos habituales del árbol rojo-negro  $x:\text{clave}$ ,  $x:\text{color}$ ,  $x:p$ ,  $x:\text{izquierda}$  y  $x:\text{derecha}$  en un nodo  $x$ , tenemos otro atributo,  $x:\text{tamaño}$ . Este atributo contiene el número de nodos (internos) en el subárbol con raíz en  $x$  (incluido el propio  $x$ ), es decir, el tamaño del subárbol. Si definimos el tamaño del centinela en 0, es decir, establecemos  $T:\text{nil}:\text{size}$  en 0, entonces tenemos la identidad

$x:\text{tamaño} \ D \ x:\text{izquierda}:\text{tamaño} \ C \ x:\text{derecha}:\text{tamaño} \ C \ 1$

No requerimos que las claves sean distintas en un árbol de estadísticas de orden. (Por ejemplo, el árbol de la figura 14.1 tiene dos claves con valor 14 y dos claves con valor 21). En presencia de claves iguales, la noción anterior de rango no está bien definida. Eliminamos esta ambigüedad para un árbol de estadísticas de orden definiendo el rango de un elemento como la posición en la que se imprimiría en un recorrido en orden del árbol. En la figura 14.1, por ejemplo, la clave 14 almacenada en un nodo negro tiene rango 5 y la clave 14 almacenada en un nodo rojo tiene rango 6.

#### Recuperar un elemento con un rango dado

Antes de mostrar cómo mantener esta información de tamaño durante la inserción y eliminación, examinemos la implementación de dos consultas de estadísticas de orden que utilizan esta información adicional. Comenzamos con una operación que recupera un elemento con un rango dado. El procedimiento  $\text{OS-SELECT}.x; i /$  devuelve un puntero al nodo que contiene la  $i$ -ésima clave más pequeña en el subárbol con raíz en  $x$ . Para encontrar el nodo  $\text{SELECT}.T:\text{root}; i /$ , clave más pequeña en un árbol estadístico llamadas  $\text{DS-}$

```

OS-SELECCIONAR.x;
i / 1 r D x:left:size C 1 2 if i ==
r 3 return x 4
elseif i<r 5 return OS-
SELECT.x:left;i/ 6
else return OS-SELECT.x:right; ir/

```

En la línea 1 de OS-SELECT, calculamos r, el rango del nodo x dentro del subárbol con raíz en x. El valor de x:left:size es el número de nodos que vienen antes de x en un recorrido de árbol en orden del subárbol con raíz en x. Por lo tanto, x:left:size C 1 es el rango de x dentro del subárbol con raíz en x. Si i D r, entonces el nodo x es el i-ésimo elemento más pequeño, por lo que devolvemos x en la línea 3. Si i<r, entonces el i-ésimo elemento más pequeño reside en el subárbol izquierdo de x, por lo que recurrimos a x:left en la línea 5. Si i>r, entonces el i-ésimo elemento más pequeño reside en el subárbol derecho de x. Dado que el subárbol con raíz en x contiene r elementos que vienen antes del subárbol derecho de x en un recorrido de árbol en orden, el i-ésimo elemento más pequeño del subárbol con raíz en x es el .ir/ éximo elemento más pequeño del subárbol con raíz en x:derecha . La línea 6 determina este elemento recursivamente.

Para ver cómo opera OS-SELECT , considere una búsqueda del 17º elemento más pequeño en el árbol de estadísticas de orden de la Figura 14.1. Comenzamos con x como raíz, cuya clave es 26, y con i D 17. Dado que el tamaño del subárbol izquierdo de 26 es 12, su rango es 13.

Por lo tanto, sabemos que el nodo con rango 17 es el 17 13 D 4to elemento más pequeño en el subárbol derecho de 26. Después de la llamada recursiva, x es el nodo con clave 41 e i D 4.

Dado que el tamaño del subárbol izquierdo de 41 es 5, su rango dentro de su subárbol es 6. Por lo tanto, sabemos que el nodo con rango 4 es el cuarto elemento más pequeño en el subárbol izquierdo de 41.

Después de la llamada recursiva, x es el nodo con clave 30, y su rango dentro de su subárbol es 2.

Por lo tanto, recurrimos una vez más para encontrar el 42 D segundo elemento más pequeño en el subárbol con raíz en el nodo con clave 38. Ahora encontramos que su subárbol izquierdo tiene tamaño 1, lo que significa que es el segundo elemento más pequeño. Así, el procedimiento devuelve un puntero al nodo con clave 38.

Debido a que cada llamada recursiva desciende un nivel en el árbol de estadísticas de orden, el tiempo total para OS-SELECT es, en el peor de los casos, proporcional a la altura del árbol. Dado que el árbol es un árbol rojo-negro, su altura es O.lg n/, donde n es el número de nodos.

Por lo tanto, el tiempo de ejecución de OS-SELECT es O.lg n/ para un conjunto dinámico de n elementos.

#### Determinar el rango de un elemento

Dado un puntero a un nodo x en un árbol estadístico de orden T, el procedimiento OS-RANK devuelve la posición de x en el orden lineal determinado por un recorrido de árbol en orden de T

```

OS-RANGO.T; x/ 1 r
D x:izquierda:tamaño C 1 2 y
D x 3 while y
  ↳ T:raíz 4 si y == 
    y:p:derecha 5 r D r C
    y:p:izquierda:tamaño C 1 6 y D y:p 7 volver r

```

El procedimiento funciona de la siguiente manera. Podemos pensar en el rango del nodo x como el número de nodos que preceden a x en un recorrido de árbol en orden, más 1 para x mismo. OS-RANK mantiene el siguiente bucle invariable:

Al comienzo de cada iteración del ciclo while de las líneas 3 a 6, r es el rango de x:key en el subárbol con raíz en el nodo y.

Usamos este bucle invariable para mostrar que OS-RANK funciona correctamente de la siguiente manera:

Inicialización: antes de la primera iteración, la línea 1 establece que r es el rango de x:key dentro del subárbol con raíz en x. Establecer y D x en la línea 2 hace que el invariante sea verdadero la primera vez que se ejecuta la prueba en la línea 3.

Mantenimiento: al final de cada iteración del ciclo while , establecemos y D y:p.

Por tanto, debemos demostrar que si r es el rango de x:key en el subárbol con raíz en y al comienzo del cuerpo del ciclo, entonces r es el rango de x:key en el subárbol con raíz en y:p al final del ciclo. cuerpo de bucle En cada iteración del ciclo while , consideramos el subárbol con raíz en y:p. Ya hemos contado el número de nodos en el subárbol con raíz en el nodo y que preceden a x en un paseo en orden, por lo que debemos sumar los nodos en el subárbol con raíz en el hermano de y que preceden a x en un paseo en orden, más 1 para y: p si también precede a x. Si y es un hijo izquierdo, entonces ni y:p ni ningún nodo en el subárbol derecho de y:p precede a x, por lo que dejamos r solo. De lo contrario, y es un hijo derecho y todos los nodos en el subárbol izquierdo de y:p preceden a x, al igual que y:p mismo.

Así, en la línea 5, sumamos y:p:left:size C 1 al valor actual de r.

Terminación: El bucle termina cuando y D T:raíz, de modo que el subárbol con raíz en y es el árbol completo. Por lo tanto, el valor de r es el rango de x:key en todo el árbol.

Como ejemplo, cuando ejecutamos OS-RANK en el árbol de estadísticas de orden de la Figura 14.1 para encontrar el rango del nodo con la clave 38, obtenemos la siguiente secuencia de valores de y:key y r en la parte superior del ciclo while :

iteración yclave r 1 38 2

2	30	4
3	41	4
4	26	17

El procedimiento devuelve el rango 17.

Dado que cada iteración del ciclo while toma  $O.1/\log n$  tiempo, y y sube un nivel en el árbol con cada iteración, el tiempo de ejecución de OS-RANK es, en el peor de los casos, proporcional a la altura del árbol:  $O.\lg n$  en un árbol estadístico de orden de  $n$  nodos.

#### Mantenimiento de los tamaños de los subárboles

Dado el atributo de tamaño en cada nodo, OS-SELECT y OS-RANK pueden calcular rápidamente información estadística de pedidos. Pero a menos que podamos mantener estos atributos de manera eficiente dentro de las operaciones básicas de modificación en los árboles rojo-negro, nuestro trabajo habrá sido en vano. Ahora mostraremos cómo mantener los tamaños de los subárboles tanto para la inserción como para la eliminación sin afectar el tiempo de ejecución asintótico de cualquiera de las operaciones.

Notamos en la Sección 13.3 que la inserción en un árbol rojo-negro consta de dos fases. La primera fase desciende por el árbol desde la raíz, insertando el nuevo nodo como hijo de un nodo existente. La segunda fase sube por el árbol, cambia de color y realiza rotaciones para mantener las propiedades rojo-negro.

Para mantener los tamaños de los subárboles en la primera fase, simplemente incrementamos `x:size` para cada nodo `x` en el camino simple atravesado desde la raíz hacia las hojas. El nuevo nodo agregado obtiene un tamaño de 1. Dado que hay  $O.\lg n$  nodos en la ruta recorrida, el costo adicional de mantener los atributos de tamaño es  $O.\lg n$ .

En la segunda fase, los únicos cambios estructurales en el árbol rojo-negro subyacente son causados por rotaciones, de las cuales hay como máximo dos. Además, una rotación es una operación local: solo dos nodos tienen sus atributos de tamaño invalidados. El enlace alrededor del cual se realiza la rotación incide sobre estos dos nodos. Refiriéndose al código para LEFT-ROTATE(`T; x`) en la Sección 13.2, agregamos las siguientes líneas:

```
13 y:tamaño D x:tamaño
14 x:tamaño D x:izquierda:tamaño C x:derecha:tamaño C 1
```

La figura 14.2 ilustra cómo se actualizan los atributos. El cambio a RIGHT ROTATE es simétrico.

Dado que se realizan como máximo dos rotaciones durante la inserción en un árbol rojo-negro, dedicamos solo  $O.1/\log n$  tiempo adicional a actualizar los atributos de tamaño en la segunda fase.

Por lo tanto, el tiempo total para la inserción en un árbol estadístico de orden de  $n$  nodos es  $O.\lg n$ , que es asintóticamente el mismo que para un árbol rojo-negro ordinario.

La eliminación de un árbol rojo-negro también consta de dos fases: la primera opera en el árbol de búsqueda subyacente y la segunda provoca como máximo tres rotaciones y, por lo demás, no realiza cambios estructurales. (Consulte la Sección 13.4.) La primera fase elimina un nodo y del árbol o lo mueve hacia arriba dentro del árbol. Para actualizar los tamaños de los subárboles, simplemente recorremos un camino simple desde el nodo y (comenzando desde su posición original dentro del árbol) hasta la raíz, disminuyendo el tamaño

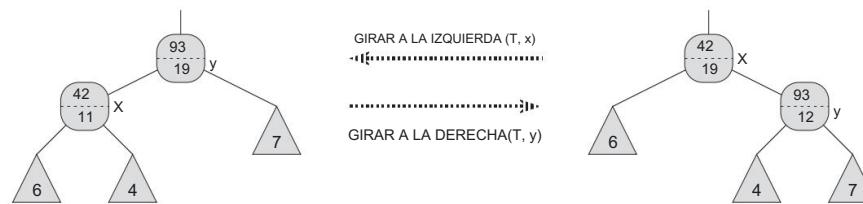


Figura 14.2 Actualización de los tamaños de los subárboles durante las rotaciones. El enlace alrededor del cual rotamos incide en los dos nodos cuyos atributos de tamaño necesitan ser actualizados. Las actualizaciones son locales y solo requieren la información de tamaño almacenada en  $x$ ,  $y$ , y las raíces de los subárboles que se muestran como triángulos.

atributo de cada nodo en la ruta. Dado que esta ruta tiene una longitud  $O.\lg n/$  en un árbol rojo-negro de  $n$  nodos, el tiempo adicional dedicado a mantener los atributos de tamaño en la primera fase es  $O.\lg n/$ . Manejamos las rotaciones  $O.1/$  en la segunda fase de eliminación de la misma manera que para la inserción. Por lo tanto, tanto la inserción como la eliminación, incluido el mantenimiento de los atributos de tamaño , requieren un tiempo  $O.\lg n/$  para un árbol de estadísticas de orden de  $n$  nodos.

### Ejercicios

#### 14.1-1

Muestre cómo OS-SELECT.T:root; 10/ opera sobre el árbol rojo-negro T de la figura 14.1.

#### 14.1-2

Muestre cómo OS-RANK.T; x/ opera sobre el árbol rojo-negro T de la figura 14.1 y el nodo x con x:clave D 35.

#### 14.1-3

Escriba una versión no recursiva de OS-SELECT.

#### 14.1-4

Escriba un procedimiento recursivo OS-KEY-RANK.T; k/ que toma como entrada un árbol estadístico de orden T y una clave k y devuelve el rango de k en el conjunto dinámico representado por T . Suponga que las claves de T son distintas.

#### 14.1-5

Dado un elemento x en un árbol estadístico de orden de  $n$  nodos y un número natural i, ¿cómo podemos determinar el i-ésimo sucesor de x en el orden lineal del árbol en  $O.\lg n/$  tiempo?

## 14.1-6

Observe que cada vez que hacemos referencia al atributo de tamaño de un nodo en OS SELECT o OS-RANK, lo usamos solo para calcular un rango. En consecuencia, supongamos que almacenamos en cada nodo su rango en el subárbol del que es raíz. mostrar cómo mantener esta información durante la inserción y eliminación. (Recuerde que estas dos operaciones pueden causar rotaciones).

## 14.1-7

Muestre cómo usar un árbol estadístico de orden para contar el número de inversiones (vea el problema 2-4) en un arreglo de tamaño  $n$  en el tiempo  $O(n \lg n)$ .

## 14.1-8 ?

Considere  $n$  cuerdas en un círculo, cada una definida por sus extremos. Describa un algoritmo  $O(n \lg n)$  para determinar el número de pares de cuerdas que se cruzan dentro del círculo. (Por ejemplo, si las  $n$  cuerdas son todas de diámetro que se encuentran en el centro, entonces la respuesta correcta es  $\frac{n(n-1)}{2}$ ). Suponga que no hay dos cuerdas que comparten un punto final.

## 14.2 Cómo aumentar una estructura de datos

El proceso de aumentar una estructura de datos básica para soportar una funcionalidad adicional ocurre con bastante frecuencia en el diseño de algoritmos. Lo usaremos nuevamente en la siguiente sección para diseñar una estructura de datos que admita operaciones en intervalos. En esta sección, examinamos los pasos involucrados en dicho aumento. También probaremos un teorema que nos permite aumentar árboles rojo-negro fácilmente en muchos casos.

Podemos dividir el proceso de aumentar una estructura de datos en cuatro pasos:

1. Elija una estructura de datos subyacente.
2. Determinar información adicional para mantener en la estructura de datos subyacente.
3. Verifique que podamos mantener la información adicional para la modificación básica operaciones sobre la estructura de datos subyacente.
4. Desarrollar nuevas operaciones.

Al igual que con cualquier método de diseño prescriptivo, no debe seguir ciegamente los pasos en el orden indicado. La mayor parte del trabajo de diseño contiene un elemento de prueba y error, y el progreso en todos los pasos generalmente se realiza en paralelo. No tiene sentido, por ejemplo, determinar información adicional y desarrollar nuevas operaciones (pasos 2 y 4) si no seremos capaces de mantener la información adicional de manera eficiente. Sin embargo, este método de cuatro pasos brinda un buen enfoque para sus esfuerzos por aumentar una estructura de datos y también es una buena forma de organizar la documentación de una estructura de datos aumentada.

Seguimos estos pasos en la Sección 14.1 para diseñar nuestros árboles de estadísticas de pedidos. Para el paso 1, elegimos árboles rojo-negro como la estructura de datos subyacente. Una pista sobre la idoneidad de los árboles rojo-negro proviene de su soporte eficiente de otras operaciones de conjuntos dinámicos en un orden total, como MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR.

Para el paso 2, agregamos el atributo de tamaño , en el que cada nodo  $x$  almacena el tamaño del subárbol con raíz en  $x$ . Generalmente, la información adicional hace que las operaciones sean más eficientes. Por ejemplo, podríamos haber implementado OS-SELECT y OS-RANK usando solo las claves almacenadas en el árbol, pero no se habrían ejecutado en  $O.\lg n / \text{time}$ .

A veces, la información adicional es información de puntero en lugar de datos, como en el ejercicio 14.2-1.

Para el paso 3, nos aseguramos de que la inserción y la eliminación pudieran mantener los atributos de tamaño mientras aún se ejecutaban en tiempo  $O.\lg n / \text{.}$  Idealmente, deberíamos necesitar actualizar solo algunos elementos de la estructura de datos para mantener la información adicional. Por ejemplo, si simplemente almacenáramos en cada nodo su rango en el árbol, los procedimientos OS-SELECT y OS-RANK se ejecutarían rápidamente, pero al insertar un nuevo elemento mínimo, esta información cambiaría en cada nodo del árbol. Cuando almacenamos tamaños de subárboles, la inserción de un nuevo elemento hace que la información cambie solo en los nodos  $O.\lg n / \text{.}$

Para el paso 4, desarrollamos las operaciones OS-SELECT y OS-RANK. Después de todo, la necesidad de nuevas operaciones es la razón por la que nos molestamos en aumentar una estructura de datos en primer lugar. Ocasionalmente, en lugar de desarrollar nuevas operaciones, usamos la información adicional para acelerar las existentes, como en el Ejercicio 14.2-1.

#### Aumento de árboles rojo-negro

Cuando los árboles rojo-negro subyacen a una estructura de datos aumentada, podemos demostrar que la inserción y la eliminación siempre pueden mantener de manera eficiente ciertos tipos de información adicional, lo que hace que el paso 3 sea muy fácil. La demostración del siguiente teorema es similar al argumento de la Sección 14.1 de que podemos mantener el atributo de tamaño para los árboles estadísticos de orden.

#### Teorema 14.1 (Aumento de un árbol rojo-negro)

Sea  $f$  un atributo que aumenta un árbol rojo-negro  $T$  de  $n$  nodos, y suponga que el valor de  $f$  para cada nodo  $x$  depende solo de la información en los nodos  $x$ ,  $x:\text{izquierda}$  y  $x:\text{derecha}$  , posiblemente incluyendo  $x : \text{izquierda}:f$  y  $x:\text{derecha}:f$  . Entonces, podemos mantener los valores de  $f$  en todos los nodos de  $T$  durante la inserción y eliminación sin afectar asintóticamente el rendimiento de  $O.\lg n / \text{ de}$  estas operaciones.

Prueba La idea principal de la prueba es que un cambio en un atributo  $f$  en un nodo  $x$  se propaga solo a los ancestros de  $x$  en el árbol. Es decir, cambiar  $x:f$  puede re-

quiere  $x:p:f$  para ser actualizado, pero nada más; la actualización de  $x:p:f$  puede requerir la actualización de  $x:p:p:f$ , pero nada más; y así sucesivamente hasta el árbol. Una vez que hayamos actualizado  $T:\text{root}:f$ , ningún otro nodo dependerá del nuevo valor, por lo que el proceso termina. Dado que la altura de un árbol rojo-negro es  $O.\lg n/$ , cambiar un atributo  $f$  en un nodo cuesta  $O.\lg n/$  tiempo para actualizar todos los nodos que dependen del cambio.

La inserción de un nodo  $x$  en  $T$  consta de dos fases. (Consulte la Sección 13.3.) La primera fase inserta  $x$  como hijo de un nodo existente  $x:p$ . Podemos calcular el valor de  $x:f$  en  $O.1/\text{tiempo}$  ya que, por suposición, depende solo de la información en los otros atributos de  $x$  mismo y la información en los hijos de  $x$ , pero los hijos de  $x$  son ambos el centinela  $T:\text{nil}$ . Una vez que hemos calculado  $x:f$ , el cambio se propaga hacia arriba en el árbol. Así, el tiempo total para la primera fase de inserción es  $O.\lg n/$ . Durante la segunda fase, los únicos cambios estructurales en el árbol provienen de las rotaciones. Dado que solo cambian dos nodos en una rotación, el tiempo total para actualizar los atributos  $f$  es  $O.\lg n/$  por rotación. Dado que el número de rotaciones durante la inserción es como máximo dos, el tiempo total para la inserción es  $O.\lg n/$ .

Al igual que la inserción, la eliminación tiene dos fases. (Consulte la Sección 13.4.) En la primera fase, los cambios en el árbol ocurren cuando el nodo eliminado se elimina del árbol. Si el nodo eliminado tenía dos hijos en ese momento, su sucesor se mueve a la posición del nodo eliminado. La propagación de las actualizaciones de  $f$  causadas por estos cambios cuesta como mucho  $O.\lg n/$ , ya que los cambios modifican el árbol localmente. Arreglar el árbol rojo-negro durante la segunda fase requiere como máximo tres rotaciones, y cada rotación requiere como máximo  $O.\lg n/$  tiempo para propagar las actualizaciones a  $f$ . Así, al igual que la inserción, el tiempo total de borrado es  $O.\lg n/$ . ■

En muchos casos, como mantener los atributos de tamaño en árboles estadísticos de orden, el costo de actualización después de una rotación es  $O.1/$ , en lugar del  $O.\lg n/$  derivado en la demostración del Teorema 14.1. El ejercicio 14.2-3 da un ejemplo.

## Ejercicios

### 14.2-1

Muestre, agregando punteros a los nodos, cómo admitir cada una de las consultas de conjunto dinámico MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR en  $O.1/\text{tiempo}$  en el peor de los casos en un árbol estadístico de orden aumentado. El rendimiento asintótico de otras operaciones en árboles de estadísticas de orden no debería verse afectado.

### 14.2-2

¿Podemos mantener las alturas negras de los nodos en un árbol rojo-negro como atributos en los nodos del árbol sin afectar el desempeño asintótico de cualquiera de las operaciones del árbol rojo-negro? Muestre cómo, o discuta por qué no. ¿Qué hay de mantener las profundidades de los nodos?

## 14.2-3 ?

Sea “ un operador binario asociativo y sea a un atributo mantenido en cada nodo de un árbol rojo-negro. Supongamos que queremos incluir en cada nodo x un atributo adicional f tal que  $x:f D x1:a " x2:a "" xm:a$ , donde  $x1; x2; \dots; xm$  es la lista ordenada de nodos en el subárbol con raíz en x. Muestre cómo actualizar los atributos f en O.1/ tiempo después de una rotación. Modifique ligeramente su argumento para aplicarlo a los atributos de tamaño en los árboles de estadísticas de orden.

## 14.2-4 ?

Deseamos aumentar los árboles rojo-negros con una operación RB ENUMERAR.x; a; b/ que genera todas las claves k tales que akb en un árbol rojo-negro con raíz en x.

Describa cómo implementar RB ENUMERAR en tiempo  $mClg n/$ , donde m es el número de claves que se generan y n es el número de nodos internos en el árbol.

(Sugerencia: no necesita agregar nuevos atributos al árbol rojo-negro).

### 14.3 Árboles de intervalo

En esta sección, aumentaremos los árboles rojo-negro para admitir operaciones en conjuntos dinámicos de intervalos. Un intervalo cerrado es un par ordenado de números reales  $t1; t2$ , con  $t1 < t2$ .

El intervalo  $t1; t2$  representa el conjunto  $t1 < t < t2$ . Los intervalos abiertos y semiabierto  $t1$  omiten ambos o uno de los puntos finales del conjunto, respectivamente. En esta sección, supondremos que los intervalos son cerrados; extender los resultados a intervalos abiertos y semiabiertos es conceptualmente sencillo.

Los intervalos son convenientes para representar eventos en los que cada uno ocupa un período continuo de tiempo. Podríamos, por ejemplo, desear consultar una base de datos de intervalos de tiempo para averiguar qué eventos ocurrieron durante un intervalo determinado. La estructura de datos de esta sección proporciona un medio eficiente para mantener dicha base de datos de intervalos.

Podemos representar un intervalo  $t1; t2$  como un objeto i, con atributos  $i:\text{low} D t1$  (el extremo inferior) e  $i:\text{high} D t2$  (el extremo superior). Decimos que los intervalos i e i0 se superponen si  $i \cap i0 \neq \emptyset$ ; es decir, si  $i:\text{bajo} i0 :alto & i0 :bajo i:\text{alto}$ . Como muestra la figura 14.3, dos intervalos i e i0 satisfacen la tricotomía del intervalo; es decir, se cumple exactamente una de las siguientes tres propiedades:

- a. i e i0 se superponen,
- b. i está a la izquierda de i0 (es decir,  $i:\text{alto} < i0 :bajo$ ),
- c. i está a la derecha de i0 (es decir,  $i0 :alto < i:\text{bajo}$ ).

Un árbol de intervalo es un árbol rojo-negro que mantiene un conjunto dinámico de elementos, con cada elemento x que contiene un intervalo  $x:int$ . Los árboles de intervalo admiten las siguientes operaciones:

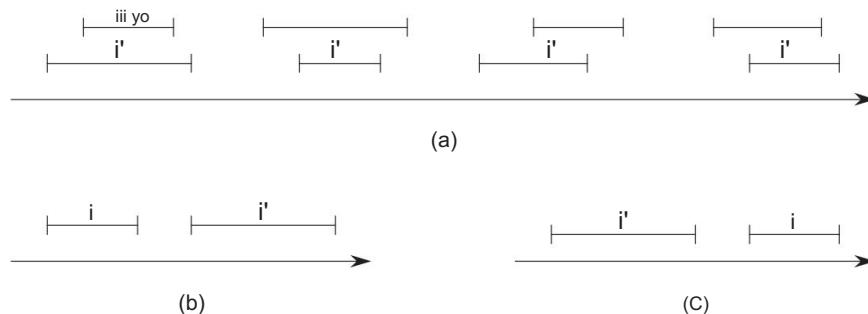


Figura 14.3 La tricotomía del intervalo para dos intervalos cerrados  $i$  e  $i_0$ . (a) Si  $i$  e  $i_0$  se superponen, hay cuatro situaciones; en cada uno,  $i$ :baja  $<$   $i_0$  :alta e  $i_0$  :baja  $<$   $i$ :alta. (b) Los intervalos no se superponen, y  $i$ :alto  $<$   $i_0$  :bajo. (c) Los intervalos no se superponen, e  $i_0$  :alto  $<$   $i$ :bajo.

**INTERVALO-INSERTAR.T;**  $x/$  añade el elemento  $x$ , cuyo atributo int se supone que contienen un intervalo, al árbol de intervalos  $T$ .

**INTERVALO-BORRAR.T;**  $x/$  elimina el elemento  $x$  del árbol de intervalos  $T$ .

**INTERVAL-SEARCH.T;**  $i/$  devuelve un puntero a un elemento  $x$  en el árbol de intervalos  $T$  tal que  $x:int$  se superpone al intervalo  $i$ , o un puntero al centinela  $T:nil$  si dicho elemento no está en el conjunto.

La figura 14.4 muestra cómo un árbol de intervalos representa un conjunto de intervalos. Realizaremos un seguimiento del método de cuatro pasos de la sección 14.2 mientras revisamos el diseño de un árbol de intervalos y las operaciones que se ejecutan en él.

#### Paso 1: estructura de datos subyacente

Elegimos un árbol rojo-negro en el que cada nodo  $x$  contiene un intervalo  $x:int$  y la clave de  $x$  es el extremo inferior,  $x:int:low$ , del intervalo. Por lo tanto, un recorrido de árbol en orden de la estructura de datos enumera los intervalos ordenados por punto final inferior.

#### Paso 2: Información adicional

Además de los propios intervalos, cada nodo  $x$  contiene un valor  $x:max$ , que es el valor máximo de cualquier extremo de intervalo almacenado en el subárbol con raíz en  $x$ .

#### Paso 3: Mantenimiento de la información

Debemos verificar que la inserción y la eliminación toman  $O.\lg n$  tiempo en un árbol de intervalos de  $n$  nodos. Podemos determinar  $x:max$  dado el intervalo  $x:int$  y los valores máximos de los hijos del nodo  $x$ :

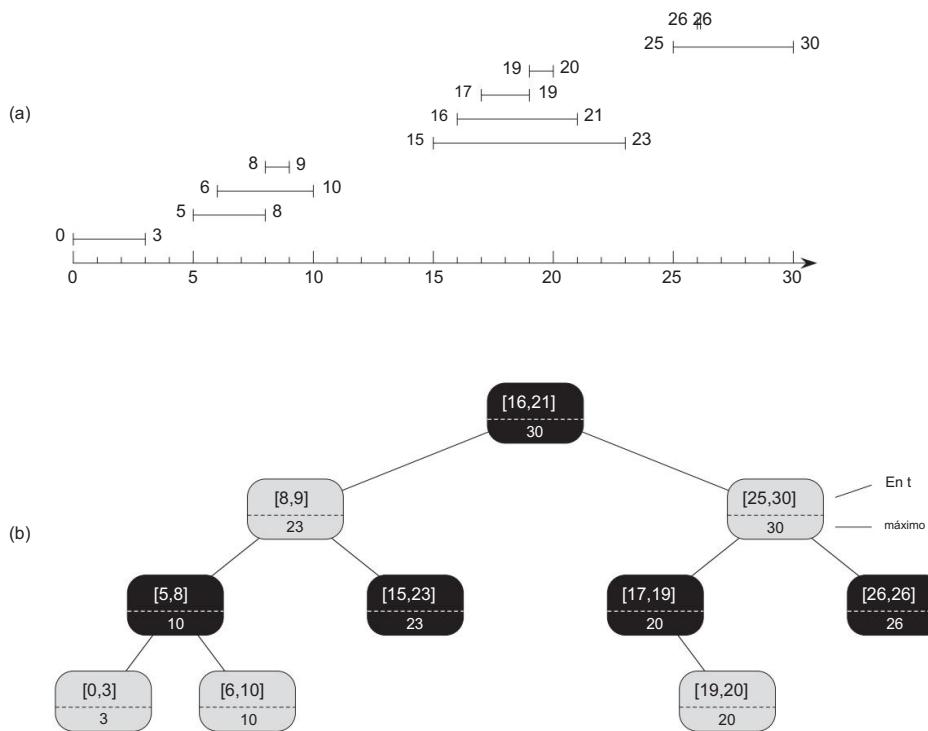


Figura 14.4 Un árbol de intervalos. (a) Un conjunto de 10 intervalos, que se muestran ordenados de abajo hacia arriba por el extremo izquierdo. (b) El árbol de intervalos que los representa. Cada nodo  $x$  contiene un intervalo, que se muestra arriba de la línea discontinua, y el valor máximo de cualquier extremo del intervalo en el subárbol con raíz en  $x$ , que se muestra debajo de la línea discontinua. Un recorrido de árbol en orden del árbol enumera los nodos ordenados por punto final izquierdo.

x:máx D máx.x:int:alto;x:izquierda:máx;x:derecha:máx/ :

Así, por el teorema 14.1, la inserción y la eliminación se realizan en tiempo  $O.\lg n/$ . De hecho, podemos actualizar los atributos máximos después de una rotación en  $O.1/$  tiempo, como muestran los ejercicios 14.2-3 y 14.3-1.

#### Paso 4: Desarrollo de nuevas operaciones

La única operación nueva que necesitamos es INTERVAL-SEARCH.T;  $i/$ , que encuentra un nodo en el árbol T cuyo intervalo se superpone al intervalo  $i$ . Si no hay ningún intervalo que se superponga a  $i$  en el árbol, el procedimiento devuelve un puntero al centinela T:nil.

```

INTERVALO-BÚSQUEDA.T;
i / 1 x D T:root 2
while x ≠ T:nil and i no superpone x:int 3 if x:left ≠ T:nil
and x:left:max i:low 4 x D x:left 5 else x D x: derecha 6
volver x

```

La búsqueda de un intervalo que se superponga a  $i$  comienza con  $x$  en la raíz del árbol y continúa hacia abajo. Termina cuando encuentra un intervalo superpuesto o  $x$  apunta al centinela  $T:nil$ . Dado que cada iteración del ciclo básico toma  $O.1/\text{tiempo}$ , y dado que la altura de un árbol rojo-negro de  $n$  nodos es  $O.\lg n/$ , el procedimiento INTERVALO-BÚSQUEDA toma  $O.\lg n/\text{tiempo}$ .

Antes de ver por qué la BÚSQUEDA EN INTERVALOS es correcta, examinemos cómo funciona en el árbol de intervalos de la figura 14.4. Supongamos que deseamos encontrar un intervalo que se superponga al intervalo  $i \in \{22, 25\}$ . Comenzamos con  $x$  como la raíz, que contiene  $\{16, 21\}$  y no se superpone  $i$ . Dado que  $x:left:max = 23$  es mayor que  $i:low = 22$ , el ciclo continúa con  $x$  como el hijo izquierdo de la raíz, el nodo que contiene  $\{8, 9\}$ , que tampoco se superpone a  $i$ . Esta vez,  $x:left:max = 10$  es menor que  $i:low = 22$ , por lo que el ciclo continúa con el hijo derecho de  $x$  como la nueva  $x$ . Porque el intervalo  $\{15, 23\}$  almacenado en este nodo se superpone a  $i$ , el procedimiento devuelve este nodo.

Como ejemplo de una búsqueda fallida, supongamos que deseamos encontrar un intervalo que se superponga  $i \in \{11, 14\}$  en el árbol de intervalos de la figura 14.4. Una vez más comenzamos con  $x$  como raíz. Dado que el intervalo de la raíz  $\{16, 21\}$  no se traslape con  $i$ , y como  $x:left:max = 23$  es mayor que  $i:low = 11$ , vamos a la izquierda hasta el nodo que contiene  $\{8, 9\}$ . Intervalo  $\{8, 9\}$  no se superpone a  $i$ , y  $x:left:max = 10$  es menor que  $i:low = 11$ , así que vamos a la derecha. (Tenga en cuenta que no hay intervalo en el subárbol izquierdo sobre las vueltas  $i$ .) Intervalo  $\{15, 23\}$  no se superpone a  $i$ , y su hijo izquierdo es  $T:nil$ , así que de nuevo vamos a la derecha, el bucle termina y devolvemos el centinela  $T:nil$ .

Para ver por qué la BÚSQUEDA EN INTERVALOS es correcta, debemos entender por qué basta con examinar una sola ruta desde la raíz. La idea básica es que en cualquier nodo  $x$ , si  $x:int$  no se superpone a  $i$ , la búsqueda siempre procede en una dirección segura: la búsqueda definitivamente encontrará un intervalo superpuesto si el árbol contiene uno. El siguiente teorema establece esta propiedad con mayor precisión.

#### Teorema 14.2

Cualquier ejecución de INTERVALO-BÚSQUEDA.T;  $i /$  devuelve un nodo cuyo intervalo se superpone a  $i$ , o bien devuelve  $T:nil$  y el árbol  $T$  no contiene ningún nodo cuyo intervalo superponga a  $i$ .

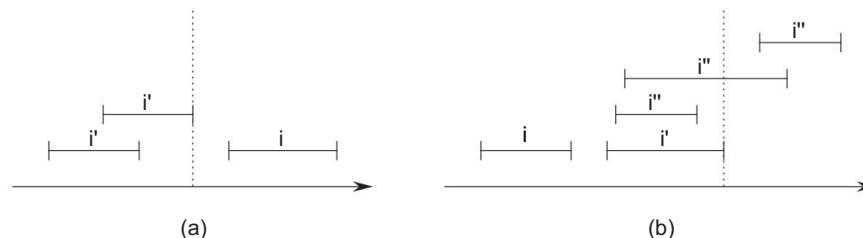


Figura 14.5 Intervalos en la demostración del Teorema 14.2. El valor de  $x:\text{left}:\text{max}$  se muestra en cada caso como una línea discontinua. (a) La búsqueda sale bien. Ningún intervalo  $i0$  en el subárbol izquierdo de  $x$  puede superponerse a  $i$ . (b) La búsqueda va hacia la izquierda. El subárbol izquierdo de  $x$  contiene un intervalo que se superpone a  $i$  (la situación no se muestra), o el subárbol izquierdo de  $x$  contiene un intervalo  $i0$  tal que  $i0:\text{high D } x:\text{left}:\text{max}$ . Como  $i$  no se superpone a  $i0$  tampoco se superpone a ningún intervalo  $i00$  en el subárbol derecho de  $x$ , ya que  $i0:\text{low } i00:\text{low}$ .

Prueba El bucle while de las líneas 2 a 5 termina cuando x D T:nil o i superpone x:int. En el último caso, ciertamente es correcto devolver x. Por lo tanto, nos enfocamos en el primer caso, en el cual el ciclo while termina porque x D T:nil.

Usamos el siguiente invariante para el bucle while de las líneas 2 a 5:

Si el árbol T contiene un intervalo que se superpone a  $i$ , entonces el subárbol con raíz en  $x$  contiene dicho intervalo.

Usamos este ciclo invariante de la siguiente manera:

Inicialización: antes de la primera iteración, la línea 1 establece que  $x$  es la raíz de  $T$  que tiene el invariante , de modo que

Mantenimiento: cada iteración del bucle while ejecuta la línea 4 o la línea 5

demonstrará que ambos casos mantienen el bucle invariante

Si se ejecuta la línea 5, entonces debido a la condición de bifurcación en la línea 3, tenemos  $x:left D T:nil$ , o  $x:left:max < i:low$ . Si  $x:left D T:nil$ , el subárbol enraizado en  $x:left$  claramente no contiene ningún intervalo que se superponga a  $i$ , por lo que establecer  $x$  en  $x:right$  mantiene el invariante. Supongamos, por tanto, que  $x:izquierda \neq T:nil$  y  $x:izquierda:máx < i:baja$ . Como muestra la figura 14.5(a), para cada intervalo  $i0$  en el subárbol izquierdo de  $x$ , tenemos

i0 : alto      x:izquierda:máx  
                < yo: bajo :

Por la tricotomía del intervalo, por lo tanto,  $i_0$  e  $i_1$  no se superponen. Por lo tanto, el subárbol izquierdo de  $x$  no contiene intervalos que se superpongan a  $i$ , por lo que establecer  $x$  en  $x:\text{right}$  mantiene la invariante.

Si, por otro lado, se ejecuta la línea 4, entonces mostraremos que se cumple la contrapositiva de la invariante del bucle. Es decir, si el subárbol con raíz en  $x:left$  no contiene ningún intervalo que se superponga a  $i$ , entonces ningún intervalo en ninguna parte del árbol se superpone a  $i$ . Dado que se ejecuta la línea 4, debido a la condición de bifurcación en la línea 3, tenemos  $x:izquierda:max \ i:baja$ . Además, por definición del atributo  $max$ , el subárbol izquierdo de  $x$  debe contener algún intervalo  $i0$  tal que

$i^0:alta \ D \ x:izquierda:máx$   
 $yo:bajo :$

(La figura 14.5(b) ilustra la situación.) Dado que  $i$  e  $i0$  no se superponen, y dado que no es cierto que  $i0:alto < i:bajo$ , por la tricotomía del intervalo se sigue que  $i:alto < i0:bajo$ . Los árboles de intervalo están codificados en los extremos inferiores de los intervalos y, por lo tanto, la propiedad del árbol de búsqueda implica que para cualquier intervalo  $i00$  en el subárbol derecho de  $x$ ,

$i:alto < i0:bajo \ i00:bajo :$

Por la tricotomía del intervalo,  $i$  e  $i00$  no se superponen. Concluimos que ya sea que algún intervalo en el subárbol izquierdo de  $x$  se superponga o no a  $i$ , establecer  $x$  en  $x:left$  mantiene el invariante.

Terminación: si el ciclo termina cuando  $x \ D \ T:nil$ , entonces el subárbol con raíz en  $x$  no contiene ningún intervalo superpuesto a  $i$ . La contrapositiva del bucle invariante implica que  $T$  no contiene ningún intervalo que se superponga a  $i$ . Por lo tanto, es correcto devolver  $x \ D \ T:nil$ .

■

Así, el procedimiento INTERVALO-BÚSQUEDA funciona correctamente.

### Ejercicios

#### 14.3-1

Escriba un pseudocódigo para LEFT-ROTATE que opere en los nodos en un árbol de intervalos y actualice los atributos máximos en O.1/ tiempo.

#### 14.3-2

Vuelva a escribir el código para BÚSQUEDA DE INTERVALOS para que funcione correctamente cuando todos los intervalos estén abiertos.

#### 14.3-3

Describa un algoritmo eficiente que, dado un intervalo  $i$ , devuelva un intervalo superpuesto a  $i$  que tenga el punto final inferior mínimo, o  $T:nil$  si no existe tal intervalo.

## 14.3-4

Dado un árbol de intervalos  $T$  y un intervalo  $i$ , describa cómo listar todos los intervalos en  $T$  que superponen  $i$  en  $O.\min.n; k \lg n//$  tiempo, donde  $k$  es el número de intervalos en la lista de salida. (Sugerencia: un método simple hace varias consultas, modificando el árbol entre consultas. Un método un poco más complicado no modifica el árbol).

## 14.3-5

Sugerir modificaciones a los procedimientos del árbol de intervalos para soportar la nueva operación INTERVALO-BÚSQUEDA-EXACTAMENTE. $T; i$ , donde  $T$  es un árbol de intervalos e  $i$  es un intervalo. La operación debe devolver un puntero a un nodo  $x$  en  $T$  tal que  $x:\text{int:low } D i:\text{low}$  y  $x:\text{int:high } D i:\text{high}$ , o  $T:\text{nil}$  si  $T$  no contiene dicho nodo.

Todas las operaciones, incluida INTERVALO-BÚSQUEDA-EXACTAMENTE, deben ejecutarse en tiempo  $O.\lg n/$  en un árbol de intervalo de  $n$  nodos.

## 14.3-6

Muestre cómo mantener un conjunto dinámico  $Q$  de números que admite la operación MIN-GAP, que da la magnitud de la diferencia de los dos números más cercanos en  $Q$ . Por ejemplo, si  $Q = \{1; 5; 9; 15; 18; 22\}$ , luego  $\text{MIN-GAP}.Q/$  devuelve  $18 - 15 = 3$ , ya que 15 y 18 son los dos números más cercanos en  $Q$ . Haga que las operaciones INSERT, DELETE, SEARCH y MIN-GAP sean lo más eficientes posible y analice su ejecución. veces.

## 14.3-7 ?

Las bases de datos VLSI comúnmente representan un circuito integrado como una lista de rectángulos. Suponga que cada rectángulo tiene una orientación rectilínea (lados paralelos a los ejes  $x$  e  $y$ ), de modo que representamos un rectángulo por sus coordenadas  $x$  e  $y$  y mínimas y máximas. Proporcione un algoritmo  $O(n \lg n)$ -time para decidir si un conjunto de  $n$  rectángulos así representados contiene o no dos rectángulos que se superponen. Su algoritmo no necesita informar todos los pares que se cruzan, pero debe informar que existe una superposición si un rectángulo cubre por completo a otro, incluso si las líneas de límite no se cruzan. (Sugerencia: mueva una línea de "barrido" a través del conjunto de rectángulos).

## Problemas

## 14-1 Punto de superposición máxima

Suponga que deseamos realizar un seguimiento de un punto de superposición máxima en un conjunto de intervalos, un punto con el mayor número de intervalos en el conjunto que lo superpone.

- Muestre que siempre habrá un punto de superposición máxima que es un punto final de uno de los segmentos.

b. Diseñe una estructura de datos que admita de manera eficiente las operaciones INTERVAL INSERT, INTERVAL-DELETE y FIND-POM, que devuelva un punto de superposición máxima. (Sugerencia: mantenga un árbol rojo-negro de todos los puntos finales. Asocie un valor de C1 con cada punto final izquierdo y asocie un valor de 1 con cada punto final derecho. Aumente cada nodo del árbol con información adicional para mantener el punto de superposición máxima).

#### 14-2 Permutación de Josefo

Definimos el problema de Josefo de la siguiente manera. Supongamos que  $n$  personas forman un círculo y que nos dan un entero positivo  $m \leq n$ . Comenzando con una primera persona designada, procedemos alrededor del círculo, eliminando a cada  $m-ésima persona. Después de eliminar a cada persona, el conteo continúa alrededor del círculo que queda. Este proceso continúa hasta que hayamos eliminado todas las  $n$  personas. El orden en el que se eliminan las personas del círculo define el  $m/n$ -Josephus permutación de los números enteros  $1; 2; \dots; n$ . Por ejemplo, el  $7/3$ -la permutación de Josefo es  $h_3; 6; 2; 7; 5; 1; 4$ .$

- a. Supongamos que  $m$  es una constante. Describa un algoritmo  $O(n)$ -time que, dado un número entero  $n$ , genere el  $m/n$ -permutación  $m$ -Josephus.
- b. Suponga que  $m$  no es una constante. Describa un algoritmo  $O(m \lg n)$ -time que, dados los enteros  $n$  y  $m$ , genere el  $m/n$ -permutación  $m$ -Josephus.

---

#### Notas del capítulo

En su libro, Preparata y Shamos [282] describen varios de los árboles de intervalo que aparecen en la literatura, citando el trabajo de H. Edelsbrunner (1980) y EM McCreight (1981). El libro detalla un árbol de intervalos que, dada una base de datos estática de  $n$  intervalos, nos permite enumerar todos los  $k$  intervalos que se superponen a un intervalo de consulta dado en el tiempo  $O(k \lg n)$ .

---

## IV Técnicas Avanzadas de Diseño y Análisis

---

## Introducción

Esta parte cubre tres técnicas importantes utilizadas en el diseño y análisis de algoritmos eficientes: programación dinámica (Capítulo 15), algoritmos codiciosos (Capítulo 16) y análisis amortizado (Capítulo 17). Partes anteriores han presentado otras técnicas ampliamente aplicables, como divide y vencerás, aleatorización y cómo resolver recurrencias. Las técnicas en esta parte son algo más sofisticadas, pero nos ayudan a atacar muchos problemas computacionales. Los temas presentados en esta parte se repetirán más adelante en este libro.

La programación dinámica generalmente se aplica a problemas de optimización en los que hacemos un conjunto de elecciones para llegar a una solución óptima. A medida que hacemos cada elección, a menudo surgen subproblemas de la misma forma. La programación dinámica es efectiva cuando un subproblema dado puede surgir de más de un conjunto parcial de opciones; la técnica clave es almacenar la solución a cada subproblema en caso de que vuelva a aparecer. El capítulo 15 muestra cómo esta simple idea a veces puede transformar algoritmos de tiempo exponencial en algoritmos de tiempo polinomial.

Al igual que los algoritmos de programación dinámica, los algoritmos codiciosos generalmente se aplican a problemas de optimización en los que hacemos un conjunto de elecciones para llegar a una solución óptima. La idea de un algoritmo codicioso es hacer cada elección de una manera localmente óptima. Un ejemplo simple es el cambio de monedas: para minimizar la cantidad de monedas estadounidenses necesarias para cambiar una cantidad determinada, podemos seleccionar repetidamente la moneda de mayor denominación que no sea mayor que la cantidad restante. Un enfoque codicioso proporciona una solución óptima para muchos de estos problemas mucho más rápido que un enfoque de programación dinámica. Sin embargo, no siempre podemos decir fácilmente si un enfoque codicioso será efectivo. El capítulo 16

teoría matroide, que proporciona una base matemática que puede ayudarnos a demostrar que un algoritmo voraz produce una solución óptima.

Usamos el análisis amortizado para analizar ciertos algoritmos que realizan una secuencia de operaciones similares. En lugar de acotar el costo de la secuencia de operaciones al acotar el costo real de cada operación por separado, un análisis amortizado proporciona un límite al costo real de toda la secuencia. Una ventaja de este enfoque es que, aunque algunas operaciones pueden ser costosas, muchas otras pueden ser económicas. En otras palabras, muchas de las operaciones podrían ejecutarse en el peor de los casos. Sin embargo, el análisis amortizado no es solo una herramienta de análisis; también es una forma de pensar sobre el diseño de algoritmos, ya que el diseño de un algoritmo y el análisis de su tiempo de ejecución suelen estar estrechamente entrelazados. El Capítulo 17 presenta tres formas de realizar un análisis amortizado de un algoritmo.

---

## 15 Programación dinámica

La programación dinámica, como el método divide y vencerás, resuelve problemas combinando las soluciones de los subproblemas. ("Programación" en este contexto se refiere a un método tabular, no a escribir código de computadora.) Como vimos en los Capítulos 2 y 4, los algoritmos divide y vencerás dividen el problema en subproblemas separados, resuelven los subproblemas recursivamente y luego combinan sus soluciones para resolver el problema original. Por el contrario, la programación dinámica se aplica cuando los subproblemas se superponen, es decir, cuando los subproblemas comparten subproblemas. En este contexto, un algoritmo divide y vencerás hace más trabajo del necesario, resolviendo repetidamente los subsubproblemas comunes. Un algoritmo de programación dinámica resuelve cada subsubproblema solo una vez y luego guarda su respuesta en una tabla, evitando así el trabajo de volver a calcular la respuesta cada vez que resuelve cada subsubproblema.

Por lo general, aplicamos la programación dinámica a los problemas de optimización. Estos problemas pueden tener muchas soluciones posibles. Cada solución tiene un valor, y deseamos encontrar una solución con el valor óptimo (mínimo o máximo). A tal solución la llamamos solución óptima del problema, en oposición a la solución óptima, ya que puede haber varias soluciones que alcancen el valor óptimo.

Al desarrollar un algoritmo de programación dinámica, seguimos una secuencia de cuatro pasos:

1. Caracterizar la estructura de una solución óptima.
2. Definir recursivamente el valor de una solución óptima.
3. Calcular el valor de una solución óptima, normalmente de forma ascendente.
4. Construya una solución óptima a partir de la información calculada.

Los pasos 1 a 3 forman la base de una solución de programación dinámica a un problema. Si solo necesitamos el valor de una solución óptima, y no la solución en sí, entonces podemos omitir el paso 4. Cuando realizamos el paso 4, a veces mantenemos información adicional durante el paso 3 para que podamos construir fácilmente una solución óptima.

Las secciones que siguen utilizan el método de programación dinámica para resolver algunos problemas de optimización. La sección 15.1 examina el problema de cortar una varilla en

varillas de menor longitud de manera que maximice su valor total. La sección 15.2 pregunta cómo podemos multiplicar una cadena de matrices mientras realizamos la menor cantidad de multiplicaciones escalares totales. Dados estos ejemplos de programación dinámica, la Sección 15.3 analiza dos características clave que debe tener un problema para que la programación dinámica sea una técnica de solución viable. La Sección 15.4 luego muestra cómo encontrar la subsecuencia común más larga de dos secuencias a través de la programación dinámica. Finalmente, la Sección 15.5 usa programación dinámica para construir árboles de búsqueda binarios que son óptimos, dada una distribución conocida de claves a buscar.

## 15.1 Corte de varillas

Nuestro primer ejemplo usa la programación dinámica para resolver un problema simple al decidir dónde cortar las varillas de acero. Serling Enterprises compra varillas largas de acero y las corta en varillas más cortas, que luego vende. Cada corte es gratis. La gerencia de Serling Enterprises quiere saber cuál es la mejor forma de cortar las varillas.

Suponemos que sabemos, para  $i \in D[1; 2; \dots]$ , el precio  $p_i$  en dólares que cobra Serling Enterprises por una varilla de longitud  $i$  pulgadas. Las longitudes de las varillas son siempre un número entero de pulgadas. La figura 15.1 da una tabla de precios de muestra.

El problema del corte de varillas es el siguiente. Dada una barra de longitud  $n$  pulgadas y una tabla de precios  $p_i$  para  $i \in D[1; 2; \dots; n]$ , determine el ingreso máximo  $r_n$  que se puede obtener cortando la varilla y vendiendo los pedazos. Tenga en cuenta que si el precio  $p_n$  de una barra de longitud  $n$  es lo suficientemente grande, es posible que una solución óptima no requiera ningún corte.

Considere el caso cuando  $n = 4$ . La figura 15.2 muestra todas las formas de cortar una barra de 4 pulgadas de largo, incluida la forma sin ningún corte. Vemos que cortar una varilla de 4 pulgadas en dos piezas de 2 pulgadas produce un ingreso  $p_2 + p_2 = 10$ , que es óptimo.

Podemos cortar una varilla de longitud  $n$  de  $2^n$  formas diferentes, ya que tenemos una opción independiente de cortar, o no cortar, a una distancia  $i$  pulgadas del extremo izquierdo,

longitud	1 2 3 4 5	6	7	8	9 10
i precio $p_i$	1 5 8 9 10 17 17 20 24 30				

Figura 15.1 Ejemplo de una tabla de precios de varillas. Cada varilla de  $i$  pulgadas de longitud genera  $p_i$  dólares de ingresos para la empresa.

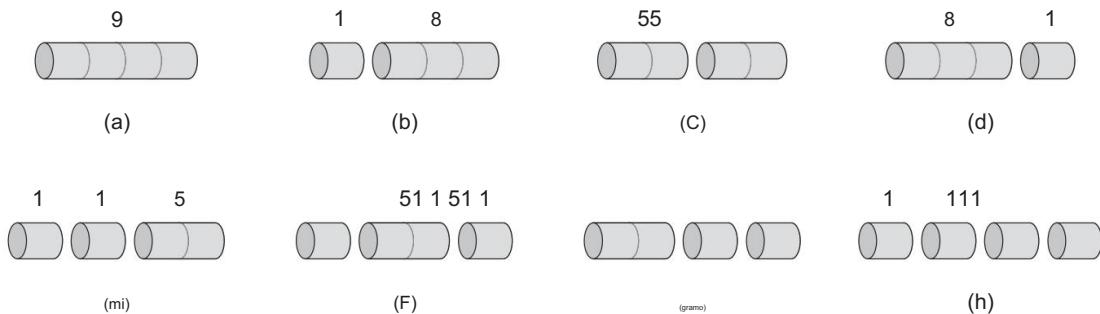


Figura 15.2 Las 8 formas posibles de cortar una barra de longitud 4. Encima de cada pieza está el valor de esa pieza, según el gráfico de precios de muestra de la Figura 15.1. La estrategia óptima es la parte (c), cortar la varilla en dos piezas de longitud 2, que tiene un valor total de 10.

para  $i \in D$ :  $1; 2; \dots; n$  <sup>1</sup> Denotamos una descomposición en partes utilizando la notación aditiva ordinaria, de modo que  $7 \in D$ :  $2 + 2 + 3$  indica que una barra de longitud 7 se corta en tres partes: dos de longitud 2 y una de longitud 3. Si una solución óptima corta la barra en  $k$  piezas, para unos  $1 \leq k \leq n$ , entonces una descomposición óptima

n D i1 C i2 CC ik

de la varilla en piezas de longitudes  $i_1, i_2, \dots, i_k$  proporciona el máximo correspondiente ganancia

rn D pi1 C pi2 CC pico :

Para nuestro problema de muestra, podemos determinar las cifras óptimas de ingresos  $r_i$ , para  $i \in D: 1; 2; \dots; 10$ , por inspección, con las correspondientes descomposiciones óptimas

Si exigieramos que las piezas se cortaran en orden de tamaño no decreciente, habría menos formas de considerar. Para  $n=4$ , consideraríamos solo 5 de estas formas: partes (a), (b), (c), (e) y (h) en la figura 15.2. El número de formas se llama función de partición; es aproximadamente igual a  $p_{2n} = 3^{np/3}$ . Esta cantidad es menor que  $2^{n^2}$ , pero aun así mucho mayor que cualquier polinomio en  $n$ .

Sin embargo, no profundizaremos más en esta línea de investigación.

r1 D 1 de la solución 1 D 1 (sin cortes) ; r2 D 5 de la  
solución 2 D 2 (sin cortes) ; r3 D 8 de la solución 3 D  
3 (sin cortes) ; r4 D 10 de la solución 4 D 2 C 2 r5 D  
13 de la solución 5 D 2 C 3 r6 D 17 de la :  
solución 6 D 6 (sin cortes) ; r7 D 18 de la :  
solución 7 D 1 C 6 o 7 D 2 C 2 C 3 r8 D 22 de la :  
solución 8 D 2 C 6 r9 D 25 de la solución 9 D 3 C 6 r10 D 30 de la :  
solución 10 D 10 (sin cortes) :  
:  
:

Más generalmente, podemos enmarcar los valores  $r_n$  para  $n \geq 1$  en términos de  $r_i$  óptimo en las varillas más cortas:

$$r_n D \max .pn; r_1 C r_1; r_2 C r_2; \dots; r_{n-1} C r_{n-1} / : \quad (15.1)$$

El primer argumento,  $p_n$ , corresponde a no hacer ningún corte y vender la barra de longitud  $n$  tal como está. Los otros  $n-1$  argumentos a  $\max$  corresponden al máximo ingreso obtenido al hacer un corte inicial de la varilla en dos piezas de tamaño  $i$  y  $n-i$ , para cada  $i \in \{1, 2, \dots, n-1\}$ , y luego cortar de manera óptima esas piezas más, obteniendo ingresos  $r_i$  y  $r_{n-i}$  de esas dos piezas. Dado que no sabemos de antemano qué valor de  $i$  optimiza los ingresos, tenemos que considerar todos los valores posibles para  $i$  y elegir el que maximiza los ingresos. También tenemos la opción de elegir ninguna  $i$  si podemos obtener más ingresos vendiendo la varilla sin cortar.

Tenga en cuenta que para resolver el problema original de tamaño  $n$ , resolvemos problemas más pequeños del mismo tipo, pero de tamaños más pequeños. Una vez que hacemos el primer corte, podemos considerar las dos piezas como instancias independientes del problema del corte de varillas. La solución óptima general incorpora soluciones óptimas a los dos subproblemas relacionados, maximizando los ingresos de cada una de esas dos piezas. Decimos que el problema del corte de varilla exhibe una subestructura óptima: las soluciones óptimas a un problema incorporan soluciones óptimas a subproblemas relacionados, que podemos resolver de forma independiente.

En una forma relacionada, pero un poco más simple, de organizar una estructura recursiva para el problema de corte de varillas, vemos una descomposición que consiste en una primera pieza de longitud  $i$  corta el extremo izquierdo, y luego un resto de longitud derecha  $y$ . Solo el resto, y no la primera parte, puede dividirse más. Podemos ver cada descomposición de una barra de longitud  $n$  de esta manera: como una primera pieza seguida de alguna descomposición del resto. Al hacerlo, podemos formular la solución sin ningún corte diciendo que la primera pieza tiene un tamaño  $i$  y un ingreso  $p_n$  y que el resto tiene un tamaño  $0$  con un ingreso correspondiente  $r_0 = 0$ . Así obtenemos la siguiente versión más simple de ecuación (15.1):

$$r_n D \max_{i=1}^{n-1} p_i C r_i / : \quad (15.2)$$

En esta formulación, una solución óptima incorpora la solución de un solo subproblema relacionado, el resto, en lugar de dos.

#### Implementación recursiva de arriba hacia abajo

El siguiente procedimiento implementa el cálculo implícito en la ecuación (15.2) de una manera directa, de arriba hacia abajo y recursiva.

```
CUT-ROD.p; n/ 1
si n == 0 2
devuelve 0 3 q D 1
4 para i D 1 a n
5 q D max.q; pŒi C
CUT-ROD.p; ni // 6 regresa q
```

El procedimiento CUT-ROD toma como entrada un arreglo  $p[1:n]$  de precios y un entero  $n$ , y devuelve el máximo ingreso posible para una barra de longitud  $n$ . Si  $n = 0$ , no es posible ningún ingreso, por lo que CUT-ROD devuelve 0 en la línea 2. La línea 3 inicializa el ingreso máximo  $q$  en 1, de modo que el ciclo for en las líneas 4 y 5 calcula correctamente  $q \leftarrow \max_{i=1}^n p_i$ . La línea 6 regresa  $q$ .

Si tuviera que codificar CUT-ROD en su lenguaje de programación favorito y ejecutarlo en su computadora, encontraría que una vez que el tamaño de entrada sea moderadamente grande, su programa tardará mucho tiempo en ejecutarse. Para  $n = 40$ , encontrará que su programa toma al menos varios minutos y muy probablemente más de una hora. De hecho, encontrará que cada vez que aumente  $n$  en 1, el tiempo de ejecución de su programa se duplicará aproximadamente.

¿Por qué CUT-ROD es tan ineficiente? El problema es que CUT-ROD se llama a sí mismo recursivamente una y otra vez con los mismos valores de parámetro; resuelve los mismos subproblemas repetidamente. La figura 15.3 ilustra lo que sucede para  $n = 4$ : CUT-ROD.p;  $n/$  llamadas CUT-ROD.p;  $ni/$  para  $i = 1, 2, 3, 4$ ; norte. De manera equivalente, CUT-ROD.p;  $n/$  llamadas CUT-ROD.p;  $j/$  para cada  $j = 0, 1, 2, 3, 4$ ;  $n/$ . Cuando este proceso se desarrolla recursivamente, la cantidad de trabajo realizado, en función de  $n$ , crece explosivamente.

Para analizar el tiempo de ejecución de CUT-ROD, sea  $T(n)$  el número total de llamadas realizadas a CUT-ROD cuando se llama con su segundo parámetro igual a  $n$ . Esta expresión es igual al número de nodos en un subárbol cuya raíz está etiquetada como  $n$  en el árbol de recursión. El recuento incluye la llamada inicial en su raíz. Así,  $T(0) = 1$  y

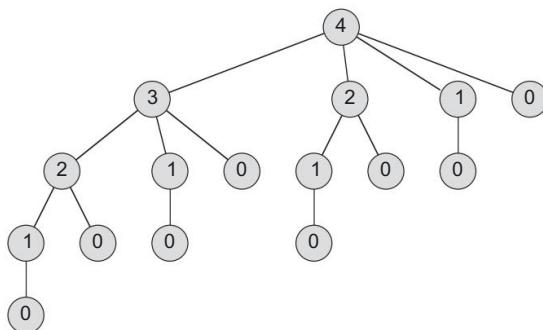


Figura 15.3 El árbol de recursividad que muestra las llamadas recursivas resultantes de una llamada  $CUT-ROD.p; n$  para  $n = 4$ . Cada etiqueta de nodo da el tamaño  $n$  del subproblema correspondiente, de modo que una arista de un parente con etiqueta  $s$  a un hijo con etiqueta  $t$  corresponde a cortar una pieza inicial de tamaño  $s$  y dejar un subproblema restante de tamaño  $t$ . Un camino desde la raíz hasta una hoja corresponde a una de las  $2n-1$  formas de cortar una varilla de longitud  $n$ . En general, este árbol de recursión tiene  $2n$  nodos y  $2n-1$  hojas.

$$T_{.n} / D_{.1} CX_{n1} T_{.j} / :_{jD0} \quad (15.3)$$

El  $1$  inicial es para la llamada en la raíz, y el término  $T_{.j} /$  cuenta el número de llamadas (incluyendo llamadas recursivas) debidas a la llamada  $CUT-ROD.p; ni /$ , donde  $j \leq D \leq n$ .

Como el Ejercicio 15.1-1 le pide que muestre,

$$T_{.n} / D_{.2n} , \dots \quad (15.4)$$

por lo que el tiempo de ejecución de  $CUT-ROD$  es exponencial en  $n$ .

En retrospectiva, este tiempo de ejecución exponencial no es tan sorprendente.  $CUT-ROD$  considera explícitamente todas las  $2n-1$  formas posibles de cortar una barra de longitud  $n$ . El árbol de llamadas recursivas tiene  $2n-1$  hojas, una para cada forma posible de cortar la barra. Las etiquetas en el camino simple desde la raíz hasta la hoja dan los tamaños de cada pieza restante de la mano derecha antes de hacer cada corte. Es decir, las etiquetas dan los puntos de corte correspondientes, medidos desde el extremo derecho de la varilla.

#### Uso de programación dinámica para un corte óptimo de varillas

Ahora mostramos cómo convertir  $CUT-ROD$  en un algoritmo eficiente, usando programación dinámica.

El método de programación dinámica funciona de la siguiente manera. Habiendo observado que una solución recursiva ingenua es ineficiente porque resuelve los mismos subproblemas repetidamente, disponemos que cada subproblema se resuelva una sola vez, guardando su solución. Si necesitamos volver a referirnos a la solución de este subproblema más adelante, podemos mirarlo

arriba, en lugar de volver a calcularlo. Por lo tanto, la programación dinámica utiliza memoria adicional para ahorrar tiempo de cálculo; sirve como ejemplo de una compensación de tiempo-memoria. Los ahorros pueden ser espectaculares: una solución de tiempo exponencial puede transformarse en una solución de tiempo polinomial. Un enfoque de programación dinámica se ejecuta en tiempo polinomial cuando el número de subproblemas distintos involucrados es polinomial en el tamaño de entrada y podemos resolver cada subproblema en tiempo polinomial.

Por lo general, hay dos formas equivalentes de implementar una programación dinámica. acercarse. Ilustraremos ambos con nuestro ejemplo de corte de varillas.

El primer enfoque es de arriba hacia abajo con memorización.<sup>2</sup> En este enfoque, escribimos el procedimiento recursivamente de manera natural, pero lo modificamos para guardar el resultado de cada subproblema (generalmente en una matriz o tabla hash). El procedimiento ahora primero verifica si ha resuelto previamente este subproblema. Si es así, devuelve el valor guardado, ahorrando más cálculos en este nivel; si no, el procedimiento calcula el valor de la manera habitual. Decimos que el procedimiento recursivo ha sido memorizado; "recuerda" qué resultados ha calculado previamente.

El segundo enfoque es el método de abajo hacia arriba. Este enfoque generalmente depende de alguna noción natural del "tamaño" de un subproblema, de modo que resolver cualquier subproblema en particular depende solo de resolver subproblemas "más pequeños". Clasificamos los subproblemas por tamaño y los resolvemos en orden de tamaño, primero el más pequeño. Al resolver un subproblema en particular, ya hemos resuelto todos los subproblemas más pequeños de los que depende su solución, y hemos guardado sus soluciones. Resolvemos cada subproblema solo una vez, y cuando lo vemos por primera vez, ya hemos resuelto todos sus subproblemas de requisitos previos.

Estos dos enfoques producen algoritmos con el mismo tiempo de ejecución asintótico, excepto en circunstancias inusuales en las que el enfoque de arriba hacia abajo en realidad no recurre para examinar todos los subproblemas posibles. El enfoque de abajo hacia arriba a menudo tiene factores constantes mucho mejores, ya que tiene menos gastos generales para las llamadas a procedimientos.

Aquí está el pseudocódigo para el procedimiento CUT-ROD de arriba hacia abajo , con memo ización agregada:

```
MEMOIZED-CUT-ROD.p; n/ 1 sea
rOE0 : : n un nuevo arreglo 2 para i D
0 to n 3 rOEi D 1 4
return MEMOIZED-CUT-
ROD-AUX.p; norte; r/
```

---

<sup>2</sup> Esto no es un error ortográfico. La palabra realmente es memorización, no memorización. Memoización viene de memo, ya que la técnica consiste en registrar un valor para que podamos consultarla más tarde.

```

MEMOIZED-CUT-ROD-AUX.p; norte; r/ 1 if
r  n 0 2 return
r  n 3 if n == 0 4 q D
0 5 else q D 1
6 for i D 1 to n 7
q D max.q; p  i C
MEMOIZED-CUT-ROD-
AUX.p; ni; r// 8 r  n D q 9 volver q

```

Aqu  , el procedimiento principal MEMOIZED-CUT-ROD inicializa un nuevo arreglo auxiliar  $r_0 : : n$  con el valor 1, una opci  n conveniente para denotar "desconocido". (Los valores de ingresos conocidos siempre son no negativos). Luego llama a su rutina auxiliar, MEMOIZED-CUT-ROD-AUX.

El procedimiento MEMOIZED-CUT-ROD-AUX es solo la versi  n memorizada de nuestro procedimiento anterior, CUT-ROD. Primero comprueba en la l  nea 1 si ya se conoce el valor deseado  $y$ , si lo es, la l  nea 2 lo devuelve. De lo contrario, las l  neas 3 a 7 calculan el valor  $q$  deseado de la manera habitual, la l  nea 8 lo guarda en  $r_n$  y la l  nea 9 lo devuelve

La versi  n de abajo hacia arriba es a  n m  s simple:

```

BOTTOM-UP-CUT-ROD.p; n/ 1 sea
r  0 : : n un nuevo arreglo 2 r  0 D
0 3 para j D 1
a n 4
      q D 1 para
5      i D 1 a jq D
6      max.q; p  i C r  j i/ r  j D q 7 8
      return r  n

```

Para el enfoque de programaci  n din  mica de abajo hacia arriba, BOTTOM-UP-CUT-ROD utiliza el orden natural de los subproblemas: un problema de tama  o  $i$  es "m  s peque  o" que un subproblema de tama  o  $j$  si  $i < j$ . As  , el procedimiento resuelve subproblemas de tama  o  $j$   $D 0; 1; : : ; n$ , en ese orden.

La l  nea 1 del procedimiento BOTTOM-UP-CUT-ROD crea una nueva matriz  $r_0 : : n$  en la que guarda los resultados de los subproblemas, y la l  nea 2 inicializa  $r_0$  a 0, ya que una barra de longitud 0 no genera ingresos. Las l  neas 3 a 6 resuelven cada subproblema de tama  o  $j$ , para  $j D 1; 2; : : ; n$ , en orden creciente de tama  o. El enfoque utilizado para resolver un problema de un tama  o particular  $j$  es el mismo que el utilizado por CUT-ROD, excepto que la l  nea 6 ahora

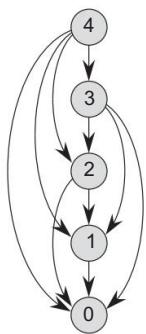


Figura 15.4 El gráfico de subproblemas para el problema de cortar varillas con  $n \leq 4$ . Las etiquetas de los vértices dan los tamaños de los subproblemas correspondientes. Un borde dirigido  $x \rightarrow y$  indica que necesitamos una solución al subproblema  $x$  para resolver el subproblema  $y$ . Este gráfico es una versión reducida del árbol de la figura 15.3, en el que todos los nodos con la misma etiqueta se colapsan en un solo vértice y todos los bordes van de padre a hijo.

hace referencia directamente a la entrada del arreglo  $rCej$  en lugar de hacer una llamada recursiva para resolver el subproblema del tamaño  $j$  en  $i$ . La línea 7 guarda en  $rCej$  la solución al subproblema de tamaño  $j$ . Finalmente, la línea 8 devuelve  $rCEn$ , que es igual al valor óptimo  $rn$ .

Las versiones de abajo hacia arriba y de arriba hacia abajo tienen el mismo tiempo de ejecución asintótico. El tiempo de ejecución del procedimiento BOTTOM-UP-CUT-ROD es  $\Theta(n^2)$ , debido a su estructura de bucle doblemente anidado. El número de iteraciones de su bucle for interno, en las líneas 5 y 6, forma una serie aritmética. El tiempo de ejecución de su contraparte de arriba hacia abajo, MEMOIZED-CUT-ROD, también es  $\Theta(n^2)$ , aunque este tiempo de ejecución puede ser un poco más difícil de ver. Debido a que una llamada recursiva para resolver un subproblema previamente resuelto regresa de inmediato, MEMOIZED-CUT-ROD resuelve cada subproblema solo una vez. Resuelve subproblemas para tamaños  $0, 1, \dots, n$ . Para resolver un subproblema de tamaño  $n$ , el bucle for de las líneas 6 y 7 itera  $n$  veces. Por lo tanto, el número total de iteraciones de este ciclo for, sobre todas las llamadas recursivas de MEMOIZED-CUT-ROD, forma una serie aritmética, dando un total de  $\Theta(n^2)$  iteraciones, al igual que el ciclo for interno de BOTTOM-UP CUT-ROD. (En realidad estamos usando una forma de análisis agregado aquí. Veremos el análisis agregado en detalle en la Sección 17.1.)

### Gráficos de subproblemas

Cuando pensamos en un problema de programación dinámica, debemos entender el conjunto de subproblemas involucrados y cómo los subproblemas dependen unos de otros.

El gráfico de subproblemas para el problema incorpora exactamente esta información. La figura 15.4 muestra el gráfico de subproblemas para el problema de cortar varillas con  $n \leq 4$ . Es un gráfico dirigido que contiene un vértice para cada subproblema distinto. el sub-

El gráfico del problema tiene un borde dirigido desde el vértice del subproblema x al vértice del subproblema y si determinar una solución óptima para el subproblema x implica considerar directamente una solución óptima para el subproblema y. Por ejemplo, el gráfico del subproblema contiene una arista de x a y si un procedimiento recursivo de arriba hacia abajo para resolver x directamente se llama a sí mismo para resolver y. Podemos pensar en el gráfico de subproblemas como una versión "reducida" o "colapsada" del árbol de recurrencia para el método recursivo de arriba hacia abajo, en el que fusionamos todos los nodos para el mismo subproblema en un solo vértice y dirigimos todos los bordes desde el padre. al niño

El método de abajo hacia arriba para la programación dinámica considera los vértices del gráfico del subproblema en un orden tal que resolvemos los subproblemas y adyacentes a un subproblema x dado antes de resolver el subproblema x. (Recuerde de la Sección B.4 que la relación de adyacencia no es necesariamente simétrica). Usando la terminología del Capítulo 22, en un algoritmo de programación dinámica de abajo hacia arriba, consideraremos los vértices del gráfico de subproblemas en un orden que es una "relación inversa". clasificación topológica", o una "clasificación topológica de la transpuesta" (ver Sección 22.4) del gráfico del subproblema. En otras palabras, no se considera ningún subproblema hasta que se hayan resuelto todos los subproblemas de los que depende. De manera similar, usando nociones del mismo capítulo, podemos ver el método de arriba hacia abajo (con memorización) para la programación dinámica como una "búsqueda primero en profundidad" del gráfico del subproblema (vea la Sección 22.3).

El tamaño del gráfico del subproblema  $G(D, V, E)$  puede ayudarnos a determinar el tiempo de ejecución del algoritmo de programación dinámica. Como resolvemos cada subproblema solo una vez, el tiempo de ejecución es la suma de los tiempos necesarios para resolver cada subproblema. Normalmente, el tiempo para calcular la solución de un subproblema es proporcional al grado (número de aristas salientes) del vértice correspondiente en el gráfico del subproblema, y el número de subproblemas es igual al número de vértices en el gráfico del subproblema. En este caso común, el tiempo de ejecución de la programación dinámica es lineal en el número de vértices y aristas.

#### Reconstruyendo una solución

Nuestras soluciones de programación dinámica para el problema del corte de varillas devuelven el valor de una solución óptima, pero no devuelven una solución real: una lista de tamaños de piezas. Podemos extender el enfoque de programación dinámica para registrar no solo el valor óptimo calculado para cada subproblema, sino también una elección que condujo al valor óptimo. Con esta información, podemos imprimir fácilmente una solución óptima.

Aquí hay una versión extendida de BOTTOM-UP-CUT-ROD que calcula, para cada tamaño de barra  $j$ , no solo el ingreso máximo  $r_j$ , sino también  $s_j$ , el tamaño óptimo de la primera pieza a cortar:

**EXTENDIDO-INFERIOR-ARRIBA-CORTE-**

VARILLA.p; n/ 1 sean r $\infty$ 0 : : n y s $\infty$ 0 : : n nuevos  
arreglos 2 r $\infty$ 0

D 0 3 para j D 1 a n 4

q D 1 para

i D 1 a j si q <

p $\infty$ i C r $\infty$ j iq D p $\infty$ i C

r $\infty$ j i s $\infty$ j D i r $\infty$ j D q

5 6 7 8

9

10 vuelta r y s

Este procedimiento es similar a BOTTOM-UP-CUT-ROD, excepto que crea la matriz s en la línea 1 y actualiza s $\infty$ j en la línea 8 para mantener el tamaño óptimo i de la primera pieza a cortar al resolver un subproblema de tamaño j.

El siguiente procedimiento toma una tabla de precios p y un tamaño de barra n, y llama EXTENDED-BOTTOM-UP-CUT-ROD para calcular la matriz s $\infty$ 1 : : n de tamaños óptimos de primera pieza y luego imprime la lista completa de tamaños de pieza en una descomposición óptima de una barra de longitud n:

**IMPRIMIR-CORTAR-VARILLA-**

SOLUCIÓN.p; n/ 1 .r; s/ D EXTENDIDO-INFERIOR-CORTE-VARILLA.p; norte/  
2 while n>0 imprime

3        s $\infty$ n n D

4        n s $\infty$ n

En nuestro ejemplo de corte de varillas, la llamada EXTENDED-BOTTOM-UP-CUT-ROD.p; 10/ devolvería las siguientes matrices:

yo 0	1 2 3 4 5 6	7      8      9 10
r $\infty$ i 0	1 5 8 10 13 17 18 22 25 30	s $\infty$ i 0 1 2 3 2 2 6 1 2 3 10

Una llamada a PRINT-CUT-ROD-SOLUTION.p; 10/ imprimiría solo 10, pero una llamada con n D 7 imprimiría los cortes 1 y 6, correspondientes a la primera descomposición óptima para r7 dada anteriormente.

**Ejercicios**

15.1-1

Demuestre que la ecuación (15.4) se sigue de la ecuación (15.3) y la condición inicial T .0/ D 1.

## 15.1-2

Muestre, por medio de un contraejemplo, que la siguiente estrategia “codiciosa” no siempre determina una forma óptima de cortar varillas. Defina la densidad de una varilla de longitud  $i$  como  $\pi_i = i$ , es decir, su valor por pulgada. La estrategia codiciosa de una barra de longitud  $n$  corta una primera pieza de longitud  $i$ , donde  $1 \leq i \leq n$ , que tiene la máxima densidad. Luego continúa aplicando la estrategia codiciosa a la pieza restante de longitud  $n-i$ .

## 15.1-3

Considere una modificación del problema del corte de varillas en el que, además del precio  $\pi_i$  por cada varilla, cada corte incurre en un costo fijo de  $c$ . El ingreso asociado a una solución es ahora la suma de los precios de las piezas menos los costos de hacer los cortes.

Proporcione un algoritmo de programación dinámica para resolver este problema modificado.

## 15.1-4

Modifique MEMOIZED-CUT-ROD para devolver no solo el valor sino también la solución real.

## 15.1-5

Los números de Fibonacci se definen por recurrencia (3.22). Proporcione un algoritmo de programación dinámica On-time para calcular el  $n$ -ésimo número de Fibonacci. Dibujar el gráfico del subproblema. ¿Cuántos vértices y aristas hay en el gráfico?

## 15.2 Multiplicación matriz-cadena

Nuestro próximo ejemplo de programación dinámica es un algoritmo que resuelve el problema de la multiplicación de matriz-cadena. Nos dan una secuencia (cadena)  $hA_1; A_2; \dots; A_n$  de  $n$  matrices a multiplicar, y deseamos calcular el producto

$$A_1 A_2 \dots A_n : \quad (15.5)$$

Podemos evaluar la expresión (15.5) usando el algoritmo estándar para multiplicar pares de matrices como una subrutina una vez que la hemos puesto entre paréntesis para resolver todas las ambigüedades sobre cómo se multiplican las matrices. La multiplicación de matrices es asociativa, por lo que todos los paréntesis dan el mismo producto. Un producto de matrices está completamente entre paréntesis si es una sola matriz o el producto de dos productos de matrices completamente entre paréntesis, entre paréntesis. Por ejemplo, si la cadena de matrices es  $hA_1; A_2; A_3; A_4$ , entonces podemos poner entre paréntesis completamente el producto  $A_1 A_2 A_3 A_4$  de cinco maneras distintas:

.A1.A2.A3A4// ; .A1..A2A3/A4// ; ..A1A2/.A3A4// ; ..A1.A2A3//A4/ ; ...A1A2/A3/A4/ :

La forma en que ponemos entre paréntesis una cadena de matrices puede tener un impacto dramático en el costo de evaluar el producto. Considere primero el costo de multiplicar dos matrices. El algoritmo estándar viene dado por el siguiente pseudocódigo, que generaliza el procedimiento SQUARE-MATRIX-MULTIPLY de la Sección 4.2. Los atributos filas y columnas son los números de filas y columnas en una matriz.

```

MATRIZ-MULTIPLICAR(A; B)
1 si A:columnas ≠ B:filas 2
error "dimensiones incompatibles"
3 si no, sea C una nueva matriz A:filas B:columnas
4     para i D 1 a A:filas
5         para j D 1 a B:columnas cij
6             D 0 para
7                 k D 1 a A:columnas cij D cij
8                     C aik bkj
9     volver C

```

Podemos multiplicar dos matrices A y B solo si son compatibles: el número de columnas de A debe ser igual al número de filas de B. Si A es una matriz apq y B es una matriz aqr, la matriz C resultante es una matriz apr. El tiempo para calcular C está dominado por el número de multiplicaciones escalares en la línea 8, que es pqr. En lo que sigue, expresaremos los costos en términos del número de multiplicaciones escalares.

Para ilustrar los diferentes costos incurridos por diferentes paréntesis de un producto matriz, considere el problema de una cadena hA1; A2; A3i de tres matrices. Suponga que las dimensiones de las matrices son 10 100, 100 5 y 5 50, respectivamente. Si multiplicamos de acuerdo con el paréntesis ..A1A2/A3/, realizamos 10 100 5 D 5000 multiplicaciones escalares para calcular el producto matricial 10 5 A1A2, más otras 10 5 50 D 2500 multiplicaciones escalares para multiplicar esta matriz por A3, para un total de 7500 multiplicaciones escalares. Si, en cambio, multiplicamos de acuerdo con el paréntesis .A1.A2A3//, realizamos 100 5 50 D 25 000 multiplicaciones escalares para calcular el producto de matriz 100 50 A2A3, más otras 10 100 50 D 50 000 multiplicaciones escalares para multiplicar A1 por esta matriz, para un total de 75.000 multiplicaciones escalares. Por lo tanto, calcular el producto según el primer paréntesis es 10 veces más rápido.

Enunciamos el problema de multiplicación de matriz-cadena como sigue: dada una cadena hA1;A2;:::;Ani de n matrices, donde para i D 1; 2; ::::; n, la matriz Ai tiene dimensión

$\pi_1 \pi_1 \dots \pi_n$ , ponga entre paréntesis completamente el producto  $A_1 A_2 \dots A_n$  de una manera que minimice el número de multiplicaciones escalares.

Tenga en cuenta que en el problema de la multiplicación de matrices en cadena, en realidad no estamos multiplicando matrices. Nuestro objetivo es solo determinar un orden para multiplicar matrices que tenga el costo más bajo. Por lo general, el tiempo invertido en determinar este orden óptimo se paga con creces con el tiempo que se ahorra más adelante al realizar las multiplicaciones de matrices (como realizar solo 7500 multiplicaciones escalares en lugar de 75 000).

#### Contar el número de paréntesis

Antes de resolver el problema de la multiplicación matriz-cadena mediante programación dinámica, convenzamos de que la verificación exhaustiva de todos los paréntesis posibles no produce un algoritmo eficiente. Denote el número de paréntesis alternativos de una secuencia de  $n$  matrices por  $P_{n/D}$ . Cuando  $n = 1$ , tenemos solo una matriz y, por lo tanto, solo una forma de poner entre paréntesis completamente el producto de matrices. Cuando  $n = 2$ , un producto de matriz completamente entre paréntesis es el producto de dos subproductos de matriz completamente entre paréntesis, y la división entre los dos subproductos puede ocurrir entre las matrices  $k$ -ésima y  $(k+1)$ -ésima para cualquier  $1 \leq k < n$ . Así, obtenemos la recurrencia

$$\begin{aligned} P_{n/D} &= 1 & \text{si } n = 1 \\ &= \sum_{k=1}^{n-1} P_{k/D} P_{(n-k)/D} & \text{si } n > 1 \end{aligned} \quad (15.6)$$

El problema 12-4 le pidió que demostrara que la solución a una recurrencia similar es la secuencia de números catalanes, que crece como  $C_n = \frac{(4n)!}{(2n+1)!(2n-1)!}$ . Un ejercicio más simple (vea el Ejercicio 15.2-3) es mostrar que la solución a la recurrencia (15.6) es  $C_n = \frac{(4n)!}{(2n+1)!(2n-1)!}$ .

El número de soluciones es, por lo tanto, exponencial en  $n$ , y el método de fuerza bruta de búsqueda exhaustiva lo convierte en una estrategia deficiente al determinar cómo poner entre paréntesis de manera óptima una cadena de matrices.

#### Aplicando programación dinámica

Usaremos el método de programación dinámica para determinar cómo poner entre paréntesis de manera óptima una cadena matricial. Al hacerlo, seguiremos la secuencia de cuatro pasos que establecimos al comienzo de este capítulo:

1. Caracterizar la estructura de una solución óptima.
2. Definir recursivamente el valor de una solución óptima.
3. Calcular el valor de una solución óptima.

4. Construya una solución óptima a partir de la información calculada.

Seguiremos estos pasos en orden, demostrando claramente cómo aplicamos cada paso al problema.

Paso 1: La estructura de un paréntesis óptimo

Para nuestro primer paso en el paradigma de programación dinámica, encontramos la subestructura óptima y luego la usamos para construir una solución óptima al problema desde soluciones óptimas hasta subproblemas. En el problema de multiplicación de matrices en cadena, podemos realizar este paso de la siguiente manera. Por donde el conveniencia, adoptemos la notación  $A_i:j$ , para la matriz que resulta de evaluar producto  $A_i A_i C_1 A_j$ . Ob  $i:j$ , sirva que si el problema no es trivial, es decir,  $i < j$ , entonces para poner entre paréntesis el producto  $A_i A_i C_1 A_j$ , debemos dividir el producto entre  $A_k$  y  $A_{k+1}$  para algún entero ~~k~~ entre  $i$  y  $j$ . Primero calculamos las matrices  $A_i:k$  y  $A_{k+1}:j$  y luego las multiplicamos para producir el producto final  $A_i:j$ . El costo de poner paréntesis de esta manera es el costo de calcular la matriz  $A_i:k$ , más el costo de calcular  $A_{k+1}:j$ , más el costo de multiplicarlos.

La subestructura óptima de este problema es la siguiente. Suponga que para poner entre paréntesis de manera óptima  $A_i A_i C_1 A_j$ , dividimos el producto entre  $A_k$  y  $A_{k+1}$ . Entonces, la forma en que ponemos entre paréntesis la subcadena "prefijo"  $A_i A_i C_1 A_k$  dentro de este paréntesis óptimo de  $A_i A_i C_1 A_j$  debe ser un paréntesis óptimo de  $A_i A_i C_1 A_k$ . ¿Por qué? Si hubiera una forma menos costosa de poner entre paréntesis  $A_i A_i C_1 A_k$ , entonces podríamos sustituir ese paréntesis en el paréntesis óptimo de  $A_i A_i C_1 A_j$  para producir otra forma de poner entre paréntesis  $A_i A_i C_1 A_j$  cuyo costo fuera menor que el óptimo: una contradicción. Una observación similar se aplica a cómo ponemos entre paréntesis la subcadena  $A_{k+1} A_{k+2} A_j$  en el paréntesis óptimo de  $A_i A_i C_1 A_j$ : debe ser un paréntesis óptimo de  $A_{k+1} A_{k+2} A_j$ .

Ahora usamos nuestra subestructura óptima para mostrar que podemos construir una solución óptima al problema a partir de soluciones óptimas a subproblemas. Hemos visto que cualquier solución a una instancia no trivial del problema de multiplicación de matrices en cadena requiere que dividamos el producto, y que cualquier solución óptima contiene soluciones óptimas para instancias de subproblemas. Por lo tanto, podemos construir una solución óptima para una instancia del problema de multiplicación de matriz-cadena dividiendo el problema en dos subproblemas (entre paréntesis de manera óptima  $A_i A_i C_1 A_k$  y  $A_{k+1} A_{k+2} A_j$ ), encontrando soluciones óptimas para instancias de subproblemas y luego combinando estas soluciones óptimas de subproblemas. Debemos asegurarnos de que cuando buscamos el lugar correcto para dividir el producto, hemos considerado todos los lugares posibles, de modo que estemos seguros de haber examinado el óptimo.

### Paso 2: una solución recursiva

A continuación, definimos recursivamente el costo de una solución óptima en términos de las soluciones óptimas a los subproblemas. Para el problema de multiplicación de cadenas de matrices, elegimos como subproblemas los problemas de determinar el costo mínimo de poner  $A_i A_i C_1 A_j$  para  $1 \leq i \leq n$  entre paréntesis  $i j n$ . Deja  $m \infty_{ij}$ ; sea el número mínimo de escalares multiplicaciones necesarias para calcular la matriz  $A_i \cdot \cdot \cdot j$ ; para el problema completo, la forma de menor costo para calcular  $A_1 \cdot \cdot \cdot n$  sería  $m \infty_{1:n}$ ; norte.

Podemos definir  $m \infty_{ij}$ ; recursivamente de la siguiente manera. Si  $i = j$ , el problema es trivial; la cadena consta de una sola matriz  $A_i \cdot \cdot \cdot i D A_i$ , por lo que no se necesitan multiplicaciones escalares para calcular el producto. Así,  $m \infty_{ii} = 0$  para  $i = 1, 2, \dots, n$ . Para calcular  $m \infty_{ij}$ ;  $j$  cuando  $i < j$ , aprovechamos la estructura de una solución óptima del paso 1. Supongamos que para poner entre paréntesis de manera óptima, dividimos el producto  $A_i A_i C_1 A_j$  entre  $A_k$  y  $A_k C_1$ , donde  $i < k < j$ . Entonces,  $m \infty_{ij}$ ;  $j$  es igual al costo mínimo para calcular los subproductos  $A_i \cdot \cdot \cdot k$  y  $A_k C_1 \cdot \cdot \cdot j$ , más el costo de multiplicar estas dos matrices juntas. Recordando que cada matriz  $A_i$  es  $p_i \times p_i$ , vemos que calcular el producto matricial  $A_i \cdot \cdot \cdot k A_k C_1 \cdot \cdot \cdot j$  requiere multiplicaciones escalares de  $p_i p_k p_j$ . Así, obtenemos

$$m \infty_{ij} = \min_{k=i+1}^{j-1} (m \infty_{ik} + m \infty_{kj}) + p_i p_k p_j$$

Esta ecuación recursiva asume que conocemos el valor de  $k$ , lo que no sabemos.

Sin embargo, sólo hay  $j-i$  valores posibles para  $k$ , a saber,  $k = i+1, i+2, \dots, j-1$ .

Dado que el paréntesis óptimo debe usar uno de estos valores para  $k$ , solo necesitamos verificarlos todos para encontrar el mejor. Por lo tanto, nuestra definición recursiva para el costo mínimo de poner entre paréntesis el producto  $A_i A_i C_1 A_j$  se convierte en

$$\min_{k=i+1}^{j-1} (m \infty_{ik} + m \infty_{kj}) + p_i p_k p_j \quad \text{si } i < j; \quad (15.7)$$

El  $m \infty_{ij}$ ; Los valores de  $j$  dan los costos de las soluciones óptimas a los subproblemas, pero no brindan toda la información que necesitamos para construir una solución óptima. Para ayudarnos a hacerlo, definimos  $s \infty_{ij}$ ; sea un valor de  $k$  en el que dividimos el producto  $A_i A_i C_1 A_j$  en un paréntesis óptimo. Es decir,  $s \infty_{ij}$ ;  $j$  es igual a un valor  $k$  tal que  $m \infty_{ij} = m \infty_{ik} + m \infty_{kj}$ .

### Paso 3: Cálculo de los costos óptimos

En este punto, podríamos escribir fácilmente un algoritmo recursivo basado en la recurrencia (15.7) para calcular el costo mínimo  $m \infty_{1:n}$  para multiplicar  $A_1 A_2 \cdots A_n$ . Como vimos para el problema del corte de varillas, y como veremos en la Sección 15.3, este algoritmo recursivo toma un tiempo exponencial, que no es mejor que el método de fuerza bruta para verificar cada forma de paréntesis del producto.

Observe que tenemos relativamente pocos subproblemas distintos: un subproblema para  $C_{i,j}$  de  $i$  y  $j$  satisface  $1 \leq i \leq n$ , o un algoritmo recursivo puede encontrar  $\frac{n(n-1)}{2} = \frac{n(n-1)}{2}$  en total. cada elección cada subproblema muchas veces en diferentes ramas de su árbol recursivo. Esta propiedad de superposición de subproblemas es el segundo sello de cuando se aplica la programación dinámica (el primer sello es la subestructura óptima).

En lugar de calcular recursivamente la solución de la recurrencia (15.7), calculamos el costo óptimo utilizando un enfoque tabular de abajo hacia arriba. (En la Sección 15.3 presentamos el enfoque de arriba hacia abajo correspondiente utilizando la memorización.)

Implementaremos el método tabular de abajo hacia arriba en el procedimiento MATRIX-CHAIN-ORDER que aparece a continuación. Este procedimiento supone que la matriz  $A_i$  tiene dimensiones  $p_i \times p_{i+1}$  para  $i = 0, 1, 2, \dots, n-1$ . Su entrada es una secuencia  $p: D(p_0, p_1, \dots, p_n)$ , donde  $p$ : longitud  $n$  y  $C[1:n]$ . El procedimiento utiliza una tabla auxiliar  $m[i][j] : : n \times n$  para almacenar el costo óptimo  $m[i][j]$ ; otra mesa auxiliar  $s[i][j] : : n \times n$  que registra qué índice de  $k$  logró el costo óptimo al calcular  $m[i][j]$ . Usaremos la tabla  $s$  para construir una solución óptima.

Para implementar el enfoque ascendente, debemos determinar a qué entradas de la tabla nos referimos al calcular  $m[i][j]$ . La ecuación (15.7) muestra que el costo  $m[i][j]$  de calcular un producto de cadena de matrices  $A_i A_{i+1} \dots A_j$  depende solo de los costos de calcular productos de cadena de matriz de menos de matrices  $A_k A_{k+1} \dots A_{j-1}$ .

Es decir, para  $k \leq i < j$ , la matriz  $A_i A_{i+1} \dots A_j$  es un producto de matrices  $A_i A_{i+1} \dots A_{j-1}$  y la matriz  $A_k A_{k+1} \dots A_{j-1}$  es un producto de matrices  $A_k A_{k+1} \dots A_{j-1}$ . Por lo tanto, el algoritmo debe completar la tabla  $m$  de manera que corresponda a resolver el problema de paréntesis en cadenas de matriz de longitud creciente. Para el subproblema de paréntesis óptimo de la cadena  $A_i A_{i+1} \dots A_j$ , consideraremos que el tamaño del subproblema es la longitud  $j - i + 1$  de la cadena.

#### MATRIZ-CADENA-ORDEN.p/ 1 n

```

D p:longitud 1 2 let m[i][j] : : n x n;
n; 1 3 para i D 1 a n-1 n y s[i][j] : : n x n; 2           n ser tablas nuevas
4 m[i][j]; i D 0 5 para l
D 2 a n // l es la longitud
de cadena 6 para i D 1 a n l C 1 7 j D i C l 1 8 m[i][j]; j D 1 9 para k
D i a j 1 10 q D m[i][j]; k C m[k] C l; j
C pi1pkpj si q < m[i][j]; j m[i][j]; j D
q s[i][j]; j re k

```

11

12

13

14 vuelta m y s

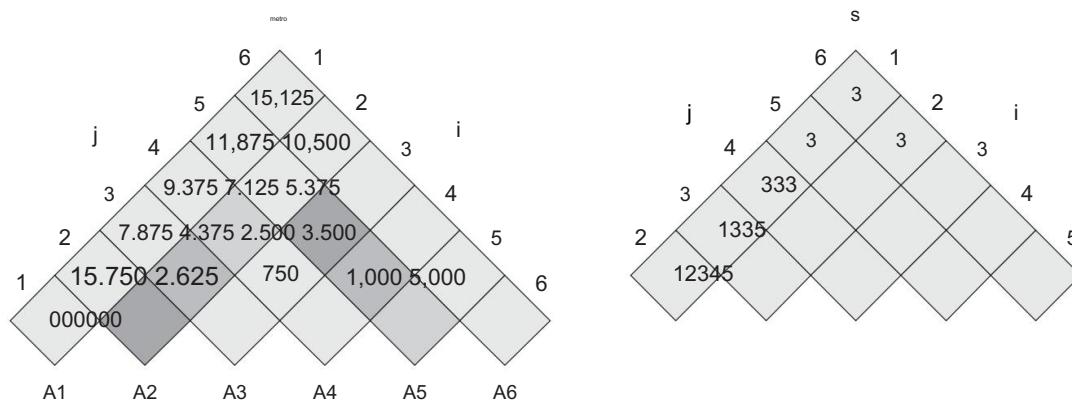


Figura 15.5 Las tablas m y s calculadas por MATRIX-CHAIN-ORDER para  $n = 6$  y las siguientes dimensiones de matriz:

matriz	A1	A2	A4	A5	A6
Dimensión A3 30 35 35 15 15 5 5 10 10 20 20 25					

Las mesas se giran para que la diagonal principal se extienda horizontalmente. La tabla m usa solo la diagonal principal y el triángulo superior, y la tabla s usa solo el triángulo superior. El número mínimo de multiplicaciones escalares para multiplicar las 6 matrices es  $m \in 1; 6 D 15.125$ . De las entradas más oscuras, los pares que tienen el mismo sombreado se juntan en la línea 10 al calcular

8 mOE2; 2 C mOE3; 5 C p1p2p5 D 0 C 2500 C 35 15 20 D 13,000 ;  
 mOE2; 5 D mín. mOE2; 3 C mOE4; 5 C p1p3p5 D 2625 C 1000 C 35 5 20 D 7125  
 ^: mOE2; 4 C mOE5; 5 C p1p4p5 D 4375 C 0 C 35 10 20 D 11,375  
 D 7125

El algoritmo primero calcula  $m \in i; 1$  para  $i = 1; 2; \dots; n$  (los costos mínimos para cadenas de longitud 1) en las líneas 3–4. Luego usa la recurrencia (15.7) para calcular  $m \in i; i + 1$  para  $i = 1; 2; \dots; n - 1$  (los costos mínimos para cadenas de longitud  $i + 1$ ) durante la primera ejecución del ciclo for en las líneas 5–13. La segunda vez que pasa por el bucle, calcula  $m \in i; i + 2$  para  $i = 1; 2; \dots; n - 2$  (los costos mínimos para cadenas de longitud  $i + 2$ ), y así sucesivamente. En cada paso, el  $m \in i; j$  el costo calculado en las líneas 10 a 13 depende únicamente de las entradas de la tabla  $m \in i; k$  y  $m \in k; j$  ya calculado.

La figura 15.5 ilustra este procedimiento en una cadena de matrices  $n = 6$ . Ya que hemos definido  $m \in i; j$  solo para  $i < j$ , solo se usa la parte de la tabla m estrictamente por encima de la diagonal principal. La figura muestra la mesa rotada para que la diagonal principal corra horizontalmente. La cadena matriz se enumera en la parte inferior. Usando este diseño, podemos encontrar el  $m \in i; j$  de costo mínimo;  $j$  para multiplicar una subcadena  $A_i A_{i+1} \dots A_j$  de matrices en la intersección de líneas que corren al noreste desde  $A_i$  y

al noroeste de  $A_j$ . Cada fila horizontal de la tabla contiene las entradas para cadenas de matriz de la misma longitud. MATRIX-CHAIN-ORDER calcula las filas de abajo hacia arriba y de izquierda a derecha dentro de cada fila. Calcula cada entrada  $m_{i,j}$  utilizando los productos  $\pi_{1 \leq k \leq j} p_k p_j$  para  $K \leq i; j \leq 1$  y todas las entradas al suroeste y sureste de  $m_{i,j}$ .

Una simple inspección de la estructura de bucle anidado de MATRIX-CHAIN-ORDER produce un tiempo de ejecución de  $O(n^3)$  para el algoritmo. Los bucles están anidados en tres profundidades, y cada índice de bucle ( $i$ ,  $j$  y  $k$ ) toma como máximo  $n$  valores. El ejercicio 15.2-5 le pide que demuestre que el tiempo de ejecución de este algoritmo es, de hecho, también  $n^3$ . El algoritmo requiere espacio  $\sim n^2$  para almacenar las tablas  $m$  y  $s$ . Por lo tanto, MATRIX-CHAIN ORDER es mucho más eficiente que el método de tiempo exponencial para enumerar todos los paréntesis posibles y verificar cada uno.

#### Paso 4: Construcción de una solución óptima

Aunque MATRIX-CHAIN-ORDER determina el número óptimo de multiplicaciones escalares necesarias para calcular un producto de matriz-cadena, no muestra directamente cómo multiplicar las matrices. La tabla  $s[1:n][1:n]$  nos da la información que hacerlo. Cada entrada  $s[i,j]$  registra un valor de  $k$  tal que un paréntesis óptimo de  $A_i A_i C_1 A_j$  divide el producto entre  $A_k$  y  $A_k C_1$ .

Por lo tanto, sabemos que la multiplicación de matrices final al calcular  $A_1:n$  de manera óptima es  $A_1:s[1:n]A_s[1:n]C_1:n$ . Podemos determinar las multiplicaciones de matrices anteriores recursivamente, ya que  $s[1:n]$  determina la última multiplicación de matrices al calcular  $A_1:s[1:n]$  y  $s[1:n]C_1:n$ . El siguiente procedimiento recursivo imprime un paréntesis óptimo de  $hA_1; A_1 C_1; \dots; A_j$ , dada la tabla  $s$  calculada por MATRIX-CHAIN ORDER y los índices  $i$  y  $j$ . La llamada inicial PRINT-OPTIMAL-PARENS. $s[1:n]$  imprime un paréntesis óptimo de  $hA_1; A_2; \dots; A_n$ .

```

PRINT-OPTIMAL-PARENS.s; i; j / 1 if
  i == j 2 print
  "A"i 3 else print "(" 4
  PRINT-OPTIMAL-
  PARENS.s; i; s[i,j] / 5 PRINT-OPTIMAL-PARENS.s;
  s[i,j] C 1; j / 6 letra ")"

```

En el ejemplo de la Figura 15.5, la llamada PRINT-OPTIMAL-PARENS. $s[1:6]$  imprime el paréntesis  $..A_1 A_2 A_3 // ..A_4 A_5 / A_6 //$ .

## Ejercicios

## 15.2-1

Encuentre un paréntesis óptimo de un producto matriz-cadena cuya secuencia de dimensiones es  $h_5; 10; 3; 12; 5; 50; 6l$ .

## 15.2-2

Proporcione un algoritmo recursivo MATRIX-CHAIN-MULTIPLY.A; s; i; j / que realmente realiza la multiplicación matriz-cadena óptima, dada la secuencia de matrices  $hA_1; A_2; \dots; A_n$ , la tabla s calculada por MATRIX-CHAIN-ORDER, y los índices i y j . (La llamada inicial sería MATRIX-CHAIN-MULTIPLY.A; s; 1; n/.)

## 15.2-3

Utilice el método de sustitución para demostrar que la solución de la recurrencia (15.6) es  $.2n!$

## 15.2-4

Describa el gráfico de subproblemas para la multiplicación de cadenas de matrices con una cadena de entrada de longitud n. ¿Cuántos vértices tiene? ¿Cuántas aristas tiene y cuáles son?

## 15.2-5

Sea  $R_i; j /$  ser el número de veces que la entrada de la tabla mce<sub>i</sub>; Se hace referencia a j mientras se calculan otras entradas de la tabla en una llamada de MATRIX-CHAIN-ORDER. Demuestre que el número total de referencias para toda la tabla es

$$\frac{X_n X_n j / d}{iD_1 j D_i} = \frac{n^3 n R_i}{3}$$

(Sugerencia: puede encontrar útil la ecuación (A.3).)

## 15.2-6

Muestre que un paréntesis completo de una expresión de n elementos tiene exactamente  $n^1$  pares de paréntesis.

## 15.3 Elementos de la programación dinámica

Aunque acabamos de trabajar con dos ejemplos del método de programación dinámica, es posible que aún se pregunte cuándo se aplica el método. Desde una perspectiva de ingeniería, ¿cuándo debemos buscar una solución de programación dinámica para un problema? En esta sección, examinamos los dos ingredientes clave que un opti-

El problema de mización debe tener para que se aplique la programación dinámica: subestructura óptima y subproblemas superpuestos. También revisamos y discutimos más a fondo cómo la memorización podría ayudarnos a aprovechar la propiedad de los subproblemas superpuestos en un enfoque recursivo de arriba hacia abajo.

### Subestructura óptima

El primer paso para resolver un problema de optimización mediante programación dinámica es caracterizar la estructura de una solución óptima. Recuerde que un problema presenta una subestructura óptima si una solución óptima del problema contiene soluciones óptimas para los subproblemas. Cada vez que un problema exhibe una subestructura óptima, tenemos una buena pista de que se puede aplicar la programación dinámica. (Sin embargo, como se analiza en el Capítulo 16, también podría significar que se aplica una estrategia codiciosa.) En la programación dinámica, construimos una solución óptima para el problema a partir de soluciones óptimas a subproblemas. En consecuencia, debemos asegurarnos de que la gama de subproblemas que consideramos incluya los que se utilizan en una solución óptima.

Descubrimos la subestructura óptima en los dos problemas que hemos examinado en este capítulo hasta ahora. En la Sección 15.1, observamos que la forma óptima de cortar una varilla de longitud  $n$  (si es que hacemos algún corte) implica cortar de manera óptima las dos piezas resultantes del primer corte. En la sección 15.2, observamos que un paréntesis óptimo de  $A_1 A_1 C_1 A_1$  que divide el producto entre  $A_k$  y  $A_k C_1$  contiene soluciones óptimas a los problemas de paréntesis de  $A_1 A_1 C_1 A_k$  y  $A_k C_1 A_k C_2 A_j$ .

Se encontrará siguiendo un patrón común en el descubrimiento de sub óptimos. estructura:

1. Muestras que una solución al problema consiste en hacer una elección, como elegir un corte inicial en una barra o elegir un índice en el que dividir la cadena de matriz. Hacer esta elección deja uno o más subproblemas por resolver.
2. Supone que para un problema dado, se le da la opción que conduce a una solución óptima. No te preocupas todavía de cómo determinar esta elección. Simplemente asumes que te lo han dado.
3. Dada esta opción, usted determina qué subproblemas surgen y cuál es la mejor manera de resolverlos. caracterizar el espacio resultante de subproblemas.
4. Usted demuestra que las soluciones a los subproblemas utilizados dentro de una solución óptima al problema deben ser óptimas mediante el uso de una técnica de "cortar y pegar". Lo hace suponiendo que cada una de las soluciones de los subproblemas no es óptima y luego derivando una contradicción. En particular, al "recortar" la solución no óptima de cada subproblema y "pegar" la óptima, demuestra que puede obtener una mejor solución al problema original, contradiciendo así su suposición de que ya tenía una solución óptima. Si un óptimo

solución da lugar a más de un subproblema, por lo general son tan similares que puede modificar el argumento de cortar y pegar para que uno se aplique a los demás con poco esfuerzo.

Para caracterizar el espacio de los subproblemas, una buena regla general dice tratar de mantener el espacio lo más simple posible y luego expandirlo según sea necesario. Por ejemplo, el espacio de subproblemas que consideramos para el problema de cortar varillas contenía los problemas de cortar óptimamente una varilla de longitud  $i$  para cada tamaño  $i$ . Este espacio de subproblemas funcionó bien y no tuvimos necesidad de probar un espacio más general de subproblemas.

Por el contrario, supongamos que hemos tratado de restringir nuestro espacio de subproblemas para la multiplicación de cadenas de matrices a productos de matrices de la forma  $A_1 A_2 A_j$ . Como antes, un paréntesis óptimo debe dividir este producto entre  $A_k$  y  $A_{k+1}$  por  $1 \leq k < j$ . A menos que podamos garantizar que  $k$  siempre es igual a  $j - 1$ , encontraríamos que tenemos subproblemas de la forma  $A_1 A_2 A_k$  y  $A_k A_{k+1} A_{k+2} A_j$ , y que el último subproblema no es de la forma  $A_1 A_2 A_j$ . Para este problema, necesitábamos permitir que nuestros subproblemas varíen en "ambos extremos", es decir, permitir que tanto  $i$  como  $j$  varíen en el subproblema  $A_i A_j A_k A_l$ .

La subestructura óptima varía según los dominios del problema de dos maneras:

1. cuántos subproblemas usa una solución óptima al problema original, y
2. cuántas opciones tenemos para determinar qué subproblema(s) usar en una solución óptima.

En el problema del corte de varillas, una solución óptima para cortar una varilla de tamaño  $n$  usa solo un subproblema (de tamaño  $n$ ), pero debemos considerar  $n$  opciones para  $i$  para determinar cuál produce una solución óptima. La multiplicación de matriz-cadena para la subcadena  $A_i A_{i+1} A_j$  sirve como ejemplo con dos subproblemas y opciones  $j-i$ . Para una matriz  $A_k$  dada en la que dividimos el producto, tenemos dos subproblemas: poner entre paréntesis  $A_i A_{i+1} A_k$  y entre paréntesis  $A_k A_{k+1} A_{k+2} A_j$ , y debemos resolver ambos de manera óptima. Una vez que determinamos las soluciones óptimas a los subproblemas, elegimos entre  $j-i$  candidatos para el índice  $k$ .

Informalmente, el tiempo de ejecución de un algoritmo de programación dinámica depende del producto de dos factores: el número total de subproblemas y cuántas opciones buscamos para cada subproblema. En el corte de varillas, teníamos  $n$  subproblemas en general y, como máximo,  $n$  opciones para examinar para cada uno, lo que arrojaba un tiempo de ejecución de  $O(n^2)$ . La multiplicación en cadena de matrices tenía  $n^2$  subproblemas en general, y en cada uno teníamos como máximo  $n$  opciones, dando un tiempo de ejecución  $O(n^3)$  (en realidad, un tiempo de ejecución  $n^{2.5}$ , según el ejercicio 15.2-5).

Por lo general, el gráfico de subproblemas ofrece una forma alternativa de realizar el mismo análisis. Cada vértice corresponde a un subproblema, y las opciones para un subproblema

problema son las aristas incidentes a ese subproblema. Recuerde que en el corte de varillas, el gráfico del subproblema tenía  $n$  vértices y, como máximo,  $n$  aristas por vértice, lo que generaba un tiempo de ejecución de  $O.n^2l$ . Para la multiplicación en cadena de matrices, si dibujáramos el gráfico del subproblema, tendría  $.n^2/$  vértices y cada vértice tendría un grado como máximo  $n$  1, dando un total de  $O.n^3/$  vértices y aristas.

La programación dinámica a menudo utiliza una subestructura óptima de forma ascendente. Es decir, primero encontramos soluciones óptimas a los subproblemas y, habiendo resuelto los subproblemas, encontramos una solución óptima al problema. Encontrar una solución óptima al problema implica hacer una elección entre los subproblemas en cuanto a cuál usaremos para resolver el problema. El costo de la solución del problema suele ser el costo del subproblema más un costo directamente atribuible a la elección misma. En el corte de varillas, por ejemplo, primero resolvemos los subproblemas de determinar formas óptimas de cortar varillas de longitud  $i$  para  $i \in 0; 1; \dots; l$ , y luego determinamos cuál de esos subproblemas arrojó una solución óptima para una barra de longitud  $n$ , usando la ecuación (15.2). El costo atribuible a la elección misma es el término  $p_i$  en la ecuación (15.2). En la multiplicación de cadenas de matrices, determinamos los paréntesis óptimos de las subcadenas de  $A_1 A_2 \dots A_l$ , y luego elegimos la matriz  $A_k$  en la cual dividir el producto. El costo atribuible a la elección en sí es el término  $p_{i+k} p_{k+j}$ .

En el Capítulo 16, examinaremos los "algoritmos voraces", que tienen muchas similitudes con la programación dinámica. En particular, los problemas a los que se aplican algoritmos voraces tienen una subestructura óptima. Una diferencia importante entre los algoritmos codiciosos y la programación dinámica es que, en lugar de encontrar primero las soluciones óptimas a los subproblemas y luego tomar una decisión informada, los algoritmos codiciosos primero hacen una elección "codicioso", la elección que se ve mejor en ese momento, y luego resuelven un problema subproblema resultante, sin molestarse en resolver todos los subproblemas menores relacionados posibles.

¡Sorprendentemente, en algunos casos esta estrategia funciona!

#### sutilezas

Debe tener cuidado de no asumir que se aplica una subestructura óptima cuando no es así. Considere los siguientes dos problemas en los que se nos da un grafo dirigido  $G = (V, E)$  y vértices  $u, v$ ; 2 voltios

Ruta más corta no ponderada:<sup>3</sup> Encuentre una ruta desde  $u$  hasta  $v$  que tenga la menor cantidad de aristas. Tal camino debe ser simple, ya que al quitar un ciclo de un camino se produce un camino con menos aristas.

---

<sup>3</sup>Usamos el término "no ponderado" para distinguir este problema del de encontrar caminos más cortos con aristas ponderadas, que veremos en los capítulos 24 y 25. Podemos usar la técnica de búsqueda primero en amplitud del capítulo 22 para resolver el problema no ponderado.

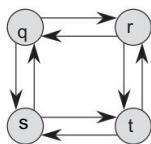


Figura 15.6 Un gráfico dirigido que muestra que el problema de encontrar un camino simple más largo en un gráfico dirigido no ponderado no tiene una subestructura óptima. El camino  $q \rightarrow r \rightarrow t$  es un camino simple más largo de  $q$  a  $t$ , pero el subcamino  $q \rightarrow r$  no es un camino simple más largo de  $q$  a  $r$ , ¡ni tampoco lo es el subcamino  $r \rightarrow t$  al camino simple más largo de  $r$  a  $t$ .

Ruta simple más larga no ponderada: busque una ruta simple desde  $u$  hasta que tenga la mayor cantidad de aristas. Necesitamos incluir el requisito de simplicidad porque, de lo contrario, podemos atravesar un ciclo tantas veces como queramos para crear caminos con un número arbitrariamente grande de aristas.

El problema del camino más corto no ponderado exhibe una subestructura óptima, como sigue. Suponga que  $u \rightarrow v$ , por lo que el problema no es trivial. Entonces, cualquier camino  $p$  desde  $u$  hasta  $v$  debe contener un vértice intermedio, digamos  $w$ . (Tenga en cuenta que  $w$  puede ser  $u$  o  $v$ .) Así, podemos descomponer el camino  $u \xrightarrow{p_0} w \xrightarrow{p_1} v \xrightarrow{p_2}$ . Claramente, el número de aristas en  $p$  es igual al número de aristas en  $p_1$  más el número de aristas en  $p_2$ . Decimos que si  $p$  es un camino óptimo (es decir, el más corto) desde  $u$  hasta  $v$ , entonces  $p_1$  debe ser el camino más corto desde  $u$  hasta  $w$ . ¿Por qué? Usamos un argumento de "cortar y pegar": si hubiera otro camino, digamos  $p_0$  de  $u$  a  $w$  con menos bordes que  $p_1$ , entonces podríamos cortar  $p_1$  y pegar en  $p_0$  con menos bordes para contradecir la optimización de  $p$ . . Simétricamente,  $p_2$  debe ser el camino más corto de  $w$  a  $v$ . Por lo tanto, podemos encontrar un camino más corto desde  $u$  hasta  $v$  considerando todos los vértices intermedios  $w$ , encontrando un vértice intermedio  $w$  que produzca el camino más corto general. En la Sección 25.2, usamos una variante de esta observación de la subestructura óptima para encontrar el camino más corto entre cada par de vértices en un gráfico dirigido ponderado.

Puede sentirse tentado a suponer que el problema de encontrar una ruta simple más larga no ponderada también presenta una subestructura óptima. Después de todo, si descomponemos un camino simple más largo  $u \xrightarrow{p_0} w \xrightarrow{p_1} v$  entonces ¿no debe ser  $p_1$  el camino simple más largo de  $u$  a  $w$ , y  $p_2$  no debe ser el camino simple más largo de  $w$  a  $v$ ? ¡La respuesta es no! La figura 15.6 proporciona un ejemplo. Considere el camino  $q \rightarrow r \rightarrow t$ , que es el camino simple más largo de  $q$  a  $t$ . es  $q \rightarrow r$  un camino simple más largo de  $q$  a  $r$ ? No, por el camino  $q \rightarrow s \rightarrow t$  es un camino simple que es más largo. es  $r \rightarrow t$  un camino simple más largo de  $r$  a  $t$ ? ¡No otra vez, para el camino  $r \rightarrow q \rightarrow s \rightarrow t$  es un camino simple que es más largo.

Este ejemplo muestra que para los caminos simples más largos, no solo el problema carece de una subestructura óptima, sino que no podemos ensamblar necesariamente una solución "legal" al problema a partir de soluciones a subproblemas. Si combinamos los caminos simples más largos  $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$ , obtenemos el camino  $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$ , que no es simple. De hecho, el problema de encontrar un camino simple más largo no ponderado no parece tener ningún tipo de subestructura óptima. Nunca se ha encontrado ningún algoritmo de programación dinámica eficiente para este problema. De hecho, este problema es NP-completo, lo que, como veremos en el capítulo 34, significa que es poco probable que encontremos una manera de resolverlo en tiempo polinomial.

¿Por qué la subestructura de un camino simple más largo es tan diferente de la de un camino más corto? Aunque una solución a un problema para los caminos más largo y más corto utiliza dos subproblemas, los subproblemas para encontrar el camino simple más largo no son independientes, mientras que para los caminos más cortos sí lo son. ¿Qué queremos decir con que los subproblemas sean independientes? Queremos decir que la solución de un subproblema no afecta la solución de otro subproblema del mismo problema. Para el ejemplo de la figura 15.6, tenemos el problema de encontrar el camino simple más largo desde  $q$  hasta  $t$  con dos subproblemas: encontrar los caminos simples más largos desde  $q$  hasta  $r$  y desde  $r$  hasta  $t$ . Para el primero de estos subproblemas, elegimos el camino  $q \rightarrow s \rightarrow t \rightarrow r$ , por lo que también hemos utilizado los vértices  $s$  y  $t$ . Ya no podemos usar estos vértices en el segundo subproblema, ya que la combinación de las dos soluciones a los subproblemas daría un camino que no es simple. Si no podemos usar el vértice  $t$  en el segundo problema, entonces no podemos resolverlo en absoluto, ya que se requiere que  $t$  esté en el camino que encontramos, y no es el vértice en el que estamos "empalmando" las soluciones del subproblema (siendo ese vértice  $r$ ). Debido a que usamos los vértices  $s$  y  $t$  en la solución de un subproblema, no podemos usarlos en la solución del otro subproblema. Sin embargo, debemos usar al menos uno de ellos para resolver el otro subproblema, y debemos usar ambos para resolverlo de manera óptima. Por lo tanto, decimos que estos subproblemas no son independientes. Visto de otra manera, el uso de recursos para resolver un subproblema (los recursos son vértices) hace que no estén disponibles para el otro subproblema.

¿Por qué, entonces, los subproblemas son independientes para encontrar el camino más corto? La respuesta es que, por naturaleza, los subproblemas no comparten recursos. Afirmamos cualquiera si un vértice  $w$  está en un camino más corto  $p$  que entonces podemos empalmar camino más corto  $u \xrightarrow{p^1} w$  y cualquier camino más corto  $w \xrightarrow{p^2} v$  para producir un camino más corto de  $u \rightarrow v$ . Estamos seguros de que, aparte de  $w$ , ningún vértice puede aparecer en ambos caminos  $p^1$  y  $p^2$ . ¿Por qué? Suponga que algún vértice  $x \neq w$  aparece tanto en  $p^1$  como en  $p^2$ , de modo que podemos descomponer  $p^1$  como  $u \xrightarrow{p_{ux}} x \xrightarrow{p_{xp}} p^2$ . Por la subestructura óptima de este problema, el camino  $p$  tiene tantas aristas como  $p^1$  y  $p^2$  juntas; vamos decir que  $p$  tiene aristas  $e$ . Ahora construyamos un camino  $p_0 \xrightarrow{p_{ux}} x \xrightarrow{p_{xp}} u$  de  $u$  a  $v$ . Debido a que hemos eliminado los caminos de  $x$  a  $w$  y de  $w$  a  $x$ , cada uno de los cuales contiene al menos un borde, el camino  $p_0$  contiene como máximo  $e - 2$  bordes, lo que contradice

la suposición de que  $p$  es el camino más corto. Por lo tanto, estamos seguros de que los subproblemas del problema del camino más corto son independientes.

Ambos problemas examinados en las Secciones 15.1 y 15.2 tienen subproblemas independientes.

En la multiplicación de cadenas de matrices, los subproblemas son las subcadenas de multiplicación  $A_1 A_2 \dots A_k$  y  $A_{k+1} A_{k+2} \dots A_j$ . Estas subcadenas son disjuntas, de modo que ninguna matriz podría incluirse en ninguna de ellas. En el corte de varillas, para determinar la mejor manera de cortar una varilla de longitud  $n$ , observamos las mejores formas de cortar varillas de longitud  $i$  para  $i = 0, 1, \dots, n-1$ . Debido a que una solución óptima al problema de longitud  $n$  incluye solo una de estas soluciones de subproblemas (después de haber cortado la primera parte), la independencia de los subproblemas no es un problema.

#### Subproblemas superpuestos

El segundo ingrediente que debe tener un problema de optimización para que se aplique la programación dinámica es que el espacio de subproblemas debe ser "pequeño" en el sentido de que un algoritmo recursivo para el problema resuelve los mismos subproblemas una y otra vez, en lugar de generar siempre nuevos subproblemas. Normalmente, el número total de subproblemas distintos es un polinomio en el tamaño de entrada. Cuando un algoritmo recursivo revisa el mismo problema repetidamente, decimos que el problema de optimización tiene subproblemas superpuestos.

<sup>4</sup> Por el contrario, un problema para el que es adecuado un enfoque de divide y vencerás generalmente genera nuevos problemas en cada paso de la recursividad. Los algoritmos de programación dinámica generalmente aprovechan los subproblemas superpuestos al resolver cada subproblema una vez y luego almacenar la solución en una tabla donde se puede consultar cuando sea necesario, utilizando un tiempo constante por búsqueda.

En la sección 15.1, examinamos brevemente cómo una solución recursiva para cortar varillas hace exponencialmente muchas llamadas para encontrar soluciones de subproblemas más pequeños. Nuestra solución de programación dinámica reduce un algoritmo recursivo de tiempo exponencial a tiempo cuadrático.

Para ilustrar la propiedad de los subproblemas superpuestos con mayor detalle, volvamos a examinar el problema de multiplicación de matrices en cadena. Volviendo a la figura 15.5, observe que MATRIX-CHAIN-ORDER busca repetidamente la solución de los subproblemas de las filas inferiores cuando resuelve los subproblemas de las filas superiores. Por ejemplo, hace referencia a la entrada  $m[0..3]; 4$  cuatro veces: durante los cálculos de  $m[0..2]; 4$ ,  $m[0..1]; 4$ ,

<sup>4</sup>Puede parecer extraño que la programación dinámica se base en que los subproblemas sean independientes y se superpongan. Aunque estos requisitos pueden sonar contradictorios, describen dos nociones diferentes, en lugar de dos puntos en el mismo eje. Dos subproblemas de un mismo problema son independientes si no comparten recursos. Dos subproblemas se superponen si en realidad son el mismo subproblema que se presenta como un subproblema de diferentes problemas.

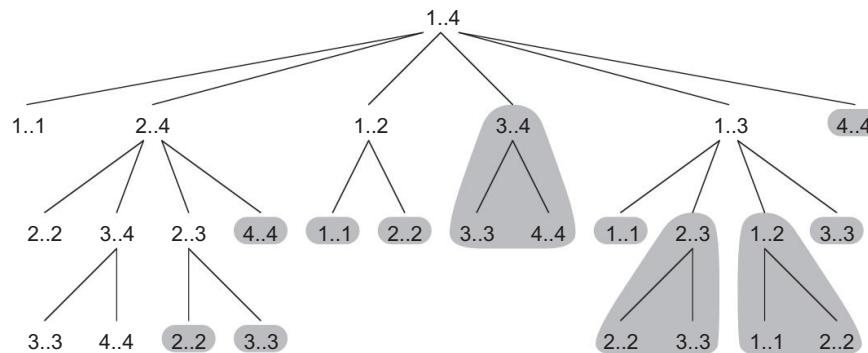


Figura 15.7 El árbol de recursión para el cálculo de RECURSIVE-MATRIX-CHAIN.p; 1; 4/. Cada nodo contiene los parámetros  $i$  y  $j$ . Los cálculos realizados en un subárbol sombreado se reemplazan por una sola tabla de búsqueda en MEMOIZED-MATRIX-CHAIN.

mOE3; 5, y mOE3; 6. Si tuviéramos que volver a calcular mOE3; 4 cada vez, en lugar de simplemente buscarlo, el tiempo de ejecución aumentaría drásticamente. Para ver cómo, considere el siguiente procedimiento recursivo (ineficiente) que determina  $mOEi; j$ , el número mínimo de multiplicaciones escalares necesarias para calcular el producto matriz-cadena  $A_i : j D A_i A_i C_1 A_j$ . El procedimiento se basa directamente en la recurrencia (15.7).

```

CADENA-MATRIZ-RECURSIVA.p; i; j / 1 si i
== j 2
devuelve 0 3 mOEi;
j D 1 4 para k D i a
j 1 5
    q D CADENA-MATRIZ-RECURSIVA.p; i; k/
    C CADENA-MATRIZ-RECURSIVA.p; k C 1; j / C
    pi1pkpj si q <
6      mOEi; j mOEi; j
    D q 7 8 volver
mOEi; j
  
```

La figura 15.7 muestra el árbol de recurrencia producido por la llamada CADENA DE MATRIZ RECURSIVA.p ; 1; 4/. Cada nodo está etiquetado por los valores de los parámetros  $i$  y  $j$ . Observe que algunos pares de valores ocurren muchas veces.

De hecho, podemos demostrar que el tiempo para calcular  $mOE1; n$  por este procedimiento recursivo es al menos exponencial en  $n$ . Sea  $T .n/$  el tiempo que tarda RECURSIVE MATRIX-CHAIN en calcular un paréntesis óptimo de una cadena de  $n$  matrices.

Debido a que la ejecución de las líneas 1 y 2 y de las líneas 6 y 7 toman al menos una unidad de tiempo, como

hace la multiplicación en la línea 5, la inspección del procedimiento produce la recurrencia

$$T .1/ \quad 1 :$$

$$T .n/ \quad 1 CXn1 .T .k/ CT .nk/ C 1/ \quad \text{para } n>1:$$

KD1

Observando que para  $i \in D[1; 2; \dots; n]$ , cada término  $T .i/$  aparece una vez como  $T .k/$  y una vez como  $T .nk/$ , y reuniendo los  $n-1$ s en la sumatoria junto con el 1 al frente, podemos reescribir la recurrencia como

$$T .n/ \underset{i \in D[1]}{2} Xn1 T .i/ C n \quad \dots \quad (15.8)$$

Probaremos que  $T .n/ \leq D .2n/$  usando el método de sustitución. Específicamente, mostraremos que  $T .n/ \leq 2n$  para todo  $n \geq 1$ . La base es fácil, ya que  $T .1/ \leq 1 \leq 2^0$ . Inductivamente, para  $n \geq 2$  tenemos

$$T .n/ \underset{i \in D[1]}{2} Xn1 \underset{i \in D[1]}{2} i C n$$

$$D2Xn2 \underset{i \in D[0]}{2} i C n$$

$$D 2.2n1 \leq C n \quad (\text{por ecuación (A.5)})$$

$$D 2n \leq C n^{2n}$$

$\vdots$

que completa la demostración. Así, la cantidad total de trabajo realizado por la llamada `RECURSIVE-MATRIX-CHAIN.p; 1; n` es al menos exponencial en  $n$ .

Compare este algoritmo recursivo de arriba hacia abajo (sin memorización) con el algoritmo de programación dinámica de abajo hacia arriba. Este último es más eficiente porque aprovecha la propiedad de los subproblemas superpuestos. La multiplicación en cadena matricial tiene solo  $n^2$  subproblemas distintos, y el algoritmo de programación dinámica resuelve cada uno exactamente una vez. El algoritmo recursivo, por otro lado, debe resolver nuevamente cada subproblema cada vez que reaparece en el árbol recursivo. Cada vez que un árbol recursivo para la solución recursiva natural de un problema contiene el mismo subproblema repetidamente, y el número total de subproblemas distintos es pequeño, la programación dinámica puede mejorar la eficiencia, a veces de manera espectacular.

### Reconstruyendo una solución óptima

Como cuestión práctica, a menudo almacenamos qué elección hicimos en cada subproblema en una tabla para que no tengamos que reconstruir esta información a partir de los costos que almacenado.

Para la multiplicación de cadenas de matrices, la tabla  $s[i][j]$  nos ahorra una cantidad significativa de trabajo al reconstruir una solución óptima. Supongamos que no mantuviéramos el  $s[i][j]$ ; tabla  $j$ , habiendo completado solo la tabla  $m[i][j]$  que contiene costos de subproblemas óptimos. Elegimos entre  $j$  posibilidades cuando determinamos qué subproblemas usar en una solución óptima para poner entre paréntesis  $A_i A_{i+1} \dots A_j$ , y  $j$  no es una constante. Por lo tanto, tomaría  $O(j)$  /  $D^{1/2}$  tiempo reconstruir qué subproblemas elegimos para solucionar un problema dado. Almacenando en  $s[i][j]$  el índice de la matriz en la que dividimos el producto  $A_i A_{i+1} \dots A_j$ , puede reconstruir cada elección en  $O(1)$  tiempo. nosotros

### Memoización

Como vimos para el problema del corte de varillas, existe un enfoque alternativo a la programación dinámica que a menudo ofrece la eficiencia del enfoque de programación dinámica de abajo hacia arriba mientras mantiene una estrategia de arriba hacia abajo. La idea es memorizar el algoritmo recursivo natural, pero ineficiente. Al igual que en el enfoque de abajo hacia arriba, mantenemos una tabla con soluciones de subproblemas, pero la estructura de control para completar la tabla se parece más al algoritmo recursivo.

Un algoritmo recursivo memorizado mantiene una entrada en una tabla para la solución de cada subproblema. Cada entrada de la tabla contiene inicialmente un valor especial para indicar que la entrada aún debe completarse. Cuando se encuentra el subproblema por primera vez a medida que se desarrolla el algoritmo recursivo, su solución se calcula y luego se almacena en la tabla.

Cada vez que nos encontramos con este subproblema, simplemente buscamos el valor almacenado en la tabla y lo devolvemos.<sup>5</sup>

Aquí hay una versión memorizada de RECURSIVE-MATRIX-CHAIN. Tenga en cuenta dónde se asemeja al método de arriba hacia abajo memorizado para el problema de corte de varillas.

<sup>5</sup>Este enfoque presupone que conocemos el conjunto de todos los parámetros posibles de los subproblemas y que hemos establecido la relación entre las posiciones de la tabla y los subproblemas. Otro enfoque, más general, es memorizar utilizando hash con los parámetros del subproblema como claves.

CADENA-MATRIZ-MEMORIZADA.p/ 1 n D

```
p:longitud 1 2 let mŒ1 :: n;
1 3 para i D 1 a n ... n ser una mesa nueva
para j D i a n 5 mŒi; j
D 1 6 return CADENA-
BUSQUEDA.m; pag; 1; norte/
```

BUSCAR-CADENA.m; pag; i; j / 1 si

```
mŒi; j < 1 2 volver
mŒi; j 3 si i == j 4 mŒi; j D
0 5 más para k
D i a j 1 6 q D LOOKUP-
CHAIN.m; pag; i; k/ C CADENA DE
BUSQUEDA.m; pag; k C 1; j / C pi1pkpj si q < mŒi; j mŒi; j
D q 9 volver mŒi; j
```

7

8

El procedimiento MEMOIZED-MATRIX-CHAIN , como MATRIX-CHAIN-ORDER, n de valores calculados mantiene una tabla mŒ1 :: n; Se ... de mŒi; j , el número mínimo necesita 1 fibra de multiplicaciones escalares para calcular la matriz Ai ::j . Cada entrada de la tabla contiene inicialmente el valor 1 para indicar que la entrada aún debe completarse. Al llamar LOOKUP-CHAIN.m; pag; i; j /, si la línea 1 encuentra que mŒi; j < 1, entonces el procedimiento simplemente devuelve el costo mŒi calculado previamente; j en la línea 2. De lo contrario, el costo se calcula como en RECURSIVE-MATRIX-CHAIN, almacenado en mŒi; j , y regresó. Por lo tanto, BUSCAR-CADENA.m; pag; i; j / siempre devuelve el valor de mŒi; j , pero lo calcula solo en la primera llamada de LOOKUP-CHAIN con estos valores específicos de i y j .

La figura 15.7 ilustra cómo MEMOIZED-MATRIX-CHAIN ahorra tiempo en comparación con RECURSIVE-MATRIX-CHAIN. Los subárboles sombreados representan valores que busca en lugar de volver a calcular.

Al igual que el algoritmo de programación dinámica de abajo hacia arriba MATRIX-CHAIN-ORDER, el procedimiento MEMOIZED-MATRIX-CHAIN se ejecuta en tiempo O.n3!. La línea 5 de MEMOIZED-MATRIX-CHAIN ejecuta ..n2/ veces. Podemos categorizar las llamadas de LOOKUP-CHAIN en dos tipos:

1. Llamadas en las que mŒi; j D 1, para que las líneas 3–9 se ejecuten, y 2.

Llamadas en las que mŒi; j < 1, por lo que LOOKUP-CHAIN simplemente regresa en la línea 2.

Hay llamadas  $.n^2$  del primer tipo, una por entrada de la tabla. Todas las llamadas del segundo tipo se realizan como llamadas recursivas por llamadas del primer tipo. Cada vez que una determinada llamada de LOOKUP-CHAIN realiza llamadas recursivas, realiza  $O(n)$  de ellas. Por lo tanto, hay llamadas  $O.n^3$  del segundo tipo en total. Cada llamada del segundo tipo toma  $O(1)$  tiempo, y cada llamada del primer tipo toma  $O(n)$  tiempo más el tiempo empleado en sus llamadas recursivas. El tiempo total, por lo tanto, es  $O.n^3$ . La memorización, por lo tanto, convierte un algoritmo de  $.n^2$ -tiempo en un algoritmo de  $O.n^3$ -tiempo.

En resumen, podemos resolver el problema de la multiplicación de cadenas de matrices mediante un algoritmo de programación dinámica memorizado de arriba hacia abajo o un algoritmo de programación dinámica de abajo hacia arriba en  $O.n^3$  tiempo. Ambos métodos aprovechan la propiedad de los subproblemas superpuestos. Solo hay  $.n^2$  subproblemas distintos en total, y cualquiera de estos métodos calcula la solución de cada subproblema solo una vez. Sin memorización, el algoritmo recursivo natural se ejecuta en tiempo exponencial, ya que los subproblemas resueltos se resuelven repetidamente.

En la práctica general, si todos los subproblemas deben resolverse al menos una vez, un algoritmo de programación dinámica de abajo hacia arriba generalmente supera al algoritmo memorizado de arriba hacia abajo correspondiente por un factor constante, porque el algoritmo de abajo hacia arriba no tiene sobrecarga para la recursividad y menos sobrecarga para el mantenimiento de la mesa. Además, para algunos problemas podemos aprovechar el patrón regular de accesos a tablas en el algoritmo de programación dinámica para reducir aún más los requisitos de tiempo o espacio. Alternativamente, si algunos subproblemas en el espacio de subproblemas no necesitan ser resueltos en absoluto, la solución memorizada tiene la ventaja de resolver solo aquellos subproblemas que definitivamente se requieren.

## Ejercicios

### 15.3-1

¿Cuál es una forma más eficiente de determinar el número óptimo de multiplicaciones en un problema de multiplicación de matriz-cadena: enumerar todas las formas de poner entre paréntesis el producto y calcular el número de multiplicaciones para cada una, o ejecutar RECURSIVE-MATRIX-CHAIN ? Justifica tu respuesta.

### 15.3-2

Dibuje el árbol recursivo para el procedimiento MERGE-SORT de la Sección 2.3.1 en un arreglo de 16 elementos. Explique por qué la memorización no logra acelerar un buen algoritmo de divide y vencerás como MERGE-SORT.

### 15.3-3

Considere una variante del problema de multiplicación de matrices en cadena en el que el objetivo es poner entre paréntesis la secuencia de matrices para maximizar, en lugar de minimizar,

el número de multiplicaciones escalares. ¿Exhibe este problema una subestructura óptima?

#### 15.3-4

Como se dijo, en la programación dinámica primero resolvemos los subproblemas y luego elegimos cuál de ellos usar en una solución óptima al problema. El profesor Capuleto afirma que no siempre necesitamos resolver todos los subproblemas para encontrar una solución óptima. Ella sugiere que podemos encontrar una solución óptima al problema de la multiplicación en cadena de matrices eligiendo siempre la matriz  $A_k$  en la cual dividir el subproducto  $A_i A_i C_1 A_j$  (al seleccionar  $k$  para minimizar la cantidad  $p_i p_k p_j$ ) antes de resolver los subproblemas. Encuentre una instancia del problema de multiplicación de matriz-cadena para el cual este enfoque codicioso produzca una solución subóptima.

#### 15.3-5

Suponga que en el problema del corte de varillas de la sección 15.1, también tuviéramos un límite  $l_i$  en el número de piezas de longitud  $i$  que podemos producir, para  $i \in D: 1; 2; \dots; l_i$ . Demuestre que la propiedad de subestructura óptima descrita en la sección 15.1 ya no se cumple.

#### 15.3-6

Imagine que desea cambiar una moneda por otra. Te das cuenta de que, en lugar de cambiar directamente una moneda por otra, es mejor que hagas una serie de transacciones a través de otras monedas y termines con la moneda que deseas. Suponga que puede operar con  $n$  monedas diferentes, numeradas  $1; 2; \dots; n$ , donde comienza con la moneda 1 y desea terminar con la moneda  $n$ . Se le da, para cada par de monedas  $i$  y  $j$ , un tipo de cambio  $r_{ij}$ , lo que significa que si comienza con  $d$  unidades de moneda  $i$ , puede cambiar por  $d r_{ij}$  unidades de moneda  $j$ .

Una secuencia de operaciones puede implicar una comisión, que depende del número de operaciones que realice. Sea  $c_k$  la comisión que se le cobra cuando realiza  $k$  transacciones. Demuestre que, si  $c_k = 0$  para todo  $k \in D: 1; 2; \dots; n$ , entonces el problema de encontrar la mejor secuencia de intercambios de la moneda 1 a la moneda  $n$  presenta una subestructura óptima. Luego demuestre que si las comisiones  $c_k$  son valores arbitrarios, entonces el problema de encontrar la mejor secuencia de intercambios de la moneda 1 a la moneda  $n$  no necesariamente exhibe una subestructura óptima.

### 15.4 Subsecuencia común más larga

Las aplicaciones biológicas a menudo necesitan comparar el ADN de dos (o más) organismos diferentes. Una hebra de ADN consiste en una cadena de moléculas llamadas

bases, donde las posibles bases son adenina, guanina, citosina y timina. Representando cada una de estas bases por su letra inicial, podemos expresar una hebra de ADN como una cadena sobre el conjunto finito fA; C; GRAMO; Tg. (Consulte el Apéndice C para ver la definición de una cadena). Por ejemplo, el ADN de un organismo puede ser S1 D ACCGGTCGAGTGC CGCGGAAGCCGGCGAA, y el ADN de otro organismo puede ser S2 D GTCGTTCGGAATGCCGTTGCTCTGTAAA. Una razón para comparar dos hebras de ADN es determinar qué tan “similares” son las dos hebras, como una medida de cuán estrechamente relacionados están los dos organismos. Podemos, y lo hacemos, definir la similitud de muchas maneras diferentes. Por ejemplo, podemos decir que dos cadenas de ADN son similares si una es una subcadena de la otra. (El Capítulo 32 explora los algoritmos para resolver este problema). En nuestro ejemplo, ni S1 ni S2 son una subcadena de la otra. Alternativamente, podríamos decir que dos hilos son similares si el número de cambios necesarios para convertir uno en el otro es pequeño. (El problema 15-5 analiza esta noción.) Otra forma más de medir la similitud de las hebras S1 y S2 es encontrar una tercera hebra S3 en la que las bases de S3 aparecen en cada uno de S1 y S2; estas bases deben aparecer en el mismo orden, pero no necesariamente en forma consecutiva. Cuanto más larga sea la hebra S3 que podamos encontrar, más similares serán S1 y S2 . En nuestro ejemplo, la hebra más larga S3 es GTCGTCGGAAGCCGGCCGAA.

Formalizamos esta última noción de similitud como el problema de la subsecuencia común más larga. Una subsecuencia de una secuencia dada es solo la secuencia dada con cero o más elementos omitidos. Formalmente, dada una secuencia XD hx1; x2;:::xmi, otra secuencia ZD h'1; '2;:::; 'ki es una subsecuencia de X si existe una secuencia estrictamente creciente hi1; i2;:::;iki de índices de X tales que para todo j D 1; 2; : : : ; k, tenemos xij D 'j . Por ejemplo, ZD hB; C; D; Bi es una subsecuencia de XD hA; B; C; B; D; A; Bi con la correspondiente secuencia índice h2; 3; 5; 7i.

Dadas dos secuencias X e Y, decimos que una secuencia Z es una subsecuencia común de X e Y si Z es una subsecuencia tanto de X como de Y. Por ejemplo, si XD hA; ANTES DE CRISTO; B;D; A; Bi y YD hB;D;C; A; B; Ai, la secuencia hB;C;Ai es . es una subsecuencia común más larga de La secuencia hB;C; Sin embargo , Ai no X e Y, ya que tiene una longitud de 3 y la secuencia hB;C; B; Ai, que también es común a ambas secuencias X e Y hB; C; B; Ai es un LCS de X e , tiene longitud 4. Y, ya que X e Y no tienen una subsecuencia , El as es la secuencia hB; D; A; Bi, común de longitud 5 o mayor.

En el problema de la subsecuencia común más larga, tenemos dos secuencias XD hx1; x2;:::xmi y YD hy1; y2;:::;yni y desea encontrar una subsecuencia común de longitud máxima de X e Y. Esta sección muestra cómo resolver eficientemente el problema LCS usando programación dinámica.

### Paso 1: Caracterización de una subsecuencia común más larga

En un enfoque de fuerza bruta para resolver el problema de LCS, enumeraríamos todas las subsecuencias de X y verificaríamos cada subsecuencia para ver si también es una subsecuencia de Y, siguiendo la pista de la subsecuencia más larga que encontraremos. Cada subsecuencia de X corresponde a un subconjunto de los índices  $f_1; 2; \dots; m$  de X. Debido a que X tiene  $2^m$  de subsecuencias, este enfoque requiere un tiempo exponencial, lo que lo hace poco práctico para secuencias largas.

Sin embargo, el problema LCS tiene una propiedad de subestructura óptima, como lo muestra el siguiente teorema. Como veremos, las clases naturales de subproblemas corresponden a pares de "prefijos" de las dos secuencias de entrada. Para ser precisos, dada una secuencia  $X D h x_1; x_2; \dots; x_m$ , definimos el  $i$ -ésimo prefijo de X, para  $i \in 0; 1; \dots; m$ , como  $X_i D h x_1; x_2; \dots; x_i$ . Por ejemplo, si  $X D h A; B; C; B; D; A; B$ , luego  $X_4 D h A; B; C; B$  y  $X_0$  es la secuencia vacía.

#### Teorema 15.1 (Subestructura óptima de un LCS)

Sea  $X D h x_1; x_2; \dots; x_m$  y  $Y D h y_1; y_2; \dots; y_n$  sean secuencias, y sean  $Z D h' z_1; z_2; \dots; z_k$  sea cualquier LCS de X e Y

1. Si  $x_m \in Z$ , entonces  $'k D x_m D y_n$  y  $Z_k$  es un LCS de  $X_m$  y  $Y_n$ .
2. Si  $x_m \notin Z$ , entonces  $'k \neq x_m$  implica que  $Z$  es un LCS de  $X_m$  e  $Y$ . Si  $x_m \notin Z$ , entonces  $'k \neq y_n$  implica que  $Z$  es un LCS de X e  $Y_n$ .

Demostración (1) Si  $'k \neq x_m$ , entonces podríamos agregar  $x_m D y_n$  a  $Z$  para obtener una subsecuencia común de X e Y de longitud  $k + 1$ , contradiciendo la suposición de que  $Z$  es la subsecuencia común más larga de X e Y. Por lo tanto, debemos tener  $'k = x_m D y_n$ .

Ahora bien, el prefijo  $Z_k$  es una subsecuencia común de longitud  $k$  de  $X_m$  y  $Y_n$ .

Queremos demostrar que es un LCS. Supongamos, con fines de contradicción, que existe una subsecuencia común W de  $X_m$  e  $Y_n$  con longitud mayor que  $k$ . Entonces, al agregar  $x_m D y_n$  a W se produce una subsecuencia común de X e Y cuya longitud es mayor que  $k$ , que es una contradicción.

(2) Si  $'k \neq x_m$ , entonces  $Z$  es una subsecuencia común de  $X_m$  e  $Y$ . Si hubiera una subsecuencia común W de  $X_m$  e  $Y$  con longitud mayor que  $k$ , entonces W también sería una subsecuencia común de X e Y, contradiciendo la suposición de que Z es un LCS de X e Y (3). La demostración es simétrica a (2). ■

La forma en que el Teorema 15.1 caracteriza las subsecuencias comunes más largas nos dice que una LCS de dos secuencias contiene dentro de ella una LCS de prefijos de las dos secuencias. Por lo tanto, el problema LCS tiene una propiedad de subestructura óptima. un recur-

La solución alternativa también tiene la propiedad de los subproblemas superpuestos, como veremos en un momento.

### Paso 2: una solución recursiva

El teorema 15.1 implica que debemos examinar uno o dos subproblemas al encontrar un LCS de  $XD h_1; x_2; \dots; x_m$  y  $YD h_1; y_2; \dots; y_n$ . Si  $x_m \neq y_n$ , debemos encontrar un LCS de  $Xm_1$  y  $Yn_1$ . Añadiendo  $x_m \neq y_n$  a este LCS se obtiene Si  $x_m \neq y_n$ , entonces LCS de  $Xm_1$  e  $Yn_1$ . Debido a que estos casos agotan todas las posibilidades, sabemos que una de las soluciones óptimas del subproblema debe aparecer dentro de un LCS de  $X$  e  $Y$ .

Podemos ver fácilmente la propiedad de los subproblemas superpuestos en el Para encontrar un LCS de  $X$  e  $Y$ , problema LCS. es posible que necesitemos encontrar  $Y$  de  $Xm_1$  e  $Yn_1$ . Los LCS de  $X$  y  $Yn_1$  y Pero cada uno de estos subproblemas tiene el subsubproblema de un LCS de  $Xm_1$  e  $Yn_1$ . Muchos otros subproblemas comparten subproblemas.

Al igual que en el problema de la multiplicación de cadenas de matrices, nuestra solución recursiva al problema LCS implica establecer una recurrencia para el valor de una solución óptima.

Definamos  $cOEi; j$  para ser la longitud de un LCS de las secuencias  $Xi$  e  $Yj$ .

Si  $i = 0$  o  $j = 0$ , una de las secuencias tiene una longitud de 0, por lo que el LCS tiene una longitud de 0. La subestructura óptima del problema LCS da la fórmula recursiva

$$\begin{aligned} cOEi; jd &= \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ \max(cOEi; j - 1; cOEi - 1; j) + 1 & \text{si } i > 0 \text{ y } j > 0 \end{cases} \\ &\quad \text{y } xi \neq yj : \end{aligned} \tag{15.9}$$

Observe que en esta formulación recursiva, una condición en el problema restringe qué subproblemas podemos considerar. Cuando  $xi \neq yj$ , podemos y debemos considerar el subproblema de encontrar un LCS de  $Xi_1$  e  $Yj_1$ . De lo contrario, consideraremos los dos subproblemas de encontrar un LCS de  $Xi$  e  $Yj_1$  y de  $Xi_1$  e  $Yj$ .

En los algoritmos de programación dinámica anteriores que hemos examinado, para el corte de varillas y la multiplicación de cadenas de matrices, no descartamos subproblemas debido a las condiciones del problema. Encontrar un LCS no es el único algoritmo de programación dinámica que descarta subproblemas en función de las condiciones del problema. Por ejemplo, el problema de la distancia de edición (vea el problema 15-5) tiene esta característica.

### Paso 3: Cálculo de la longitud de un LCS

Con base en la ecuación (15.9), podríamos escribir fácilmente un algoritmo recursivo de tiempo exponencial para calcular la longitud de un LCS de dos secuencias. Dado que el problema LCS

tiene solo  $m/n$  subproblemas distintos, sin embargo, podemos usar la programación dinámica para calcular las soluciones de abajo hacia arriba.

El procedimiento LCS-LENGTH toma dos secuencias  $XD$   $x_1; x_2; \dots; x_m$  y  $YD$   $y_1; y_2; \dots; y_n$  como entradas. Almacena el  $c[i;j]$  valores en una tabla  $c[0:m][0:n]$ , y calcula las entradas en orden de fila principal. (Es decir, el procedimiento llena la primera fila de  $c$  de izquierda a derecha, luego la segunda fila y así sucesivamente.) El procedimiento también mantiene la tabla  $b[0:m][0:n]$  para ayudarnos a construir una solución óptima. Intuitivamente,  $b[i;j]$  apunta a la entrada de la tabla correspondiente a la solución óptima del subproblema elegida al calcular  $c[i;j]$ . El procedimiento devuelve las tablas  $b$  y  $c$ ;  $c[m][n]$  contiene la longitud de un LCS de  $X$  e  $Y$ .

#### LCS-LONGITUD.X; Y / 1

```

m D X:longitud 2 n D
Y:longitud 3 sea
b[0:m][0:n] 1 4 para i = n y c[0:m][0:n] 1 n ser tablas nuevas
i D 1 a m 5 c[i;j] 0 D 0
6 para j D 0 a n c[0:m][0:n] 0;
j D 0 7

8 para i D 1 a m
9      para j D 1 an si xi
10         == yj c[i;j];
11         j D c[i;j] 1; j 1 C 1 b[i;j]; j D "-" elseif
12         c[i;j] 1; jc[i;j] 1
13         c[i;j] D c[i;j] 1; j b[i;j]; j D "" else
14         c[i;j] D c[i;j] 1 b[i;j]; j
15         D 17 18 return c
16         y b

```

La Figura 15.8 muestra las tablas producidas por LCS-LENGTH sobre las secuencias  $XD$   $hA; B; C; B; D; A; Bi$  y  $YD$   $hB; D; C; A; B; Ai$ . El tiempo de ejecución del procedimiento es  $m \cdot n$ , ya que cada entrada de la tabla toma  $1/n$  de tiempo para calcularse.

#### Paso 4: Construcción de un LCS

La tabla  $b$  devuelta por LCS-LENGTH nos permite construir rápidamente un LCS de  $XD$   $x_1; x_2; \dots; x_m$  y  $YD$   $y_1; y_2; \dots; y_n$ . Simplemente comenzamos en  $b[m][n]$  y trace a través de la tabla siguiendo las flechas. Cada vez que encontramos un  $"."$  en la entrada  $b[i;j]$ , implica que  $x_i$  y  $y_j$  es un elemento del LCS que LCS-LENGTH

	j	0 1 2 3 4 5 6
i	yj	B D C A B A
0	xi	0 0 0 0 0 0 0
1	A	0 0 0 0 1
2	B	0 1 ← 1 ← 1 1 2 2 ←
3	C	0 1 ↑ 1 2 2 ← 2 2 2 ↑
4	B	0 1 ↑ 1 2 2 ↑ ↑ 3 ← 3
5	D	0 1 2 2 2 3 3 ↑ ↑ ↑ ↑
6	A	0 1 2 2 3 3 ↑ ↑ ↑ ↑ 4
7	B	0 1 2 2 3 4 4 ↑ ↑ ↑ ↑

Figura 15.8 Las tablas c y b calculadas por LCS-LENGTH en las secuencias XD hA; B; C; B; D; A; Bi y YD hB; D; C; A; B; Ai. El cuadrado de la fila i y la columna j contiene el valor de  $c_{i,j}$ ; j y la flecha correspondiente al valor de  $b_{i,j}$ ; j. La entrada 4 en  $c_{7,7}$ ; 6, la esquina inferior derecha de la tabla, es la longitud de un LCS hB; C; B; Ai de X e Y. para yo; j > 0, entrada  $c_{i,j}$ ; j depende solo de si  $x_i$  D y los valores en las entradas  $c_{i-1,j-1}$ ; j,  $c_{i-1,j}$  y  $c_{i,j-1}$ ; j 1, que se calculan antes de  $c_{i,j}$ ; j. Para reconstruir los elementos de un LCS, siga el  $b_{i,j}$ ; flechas j desde la esquina inferior derecha; la secuencia está sombreada. Cada “-” en la secuencia sombreada corresponde a una entrada (resaltada) para la cual  $x_i$  D yj es miembro de un LCS.

encontró. Con este método, encontramos los elementos de este LCS en orden inverso. El siguiente procedimiento recursivo imprime un LCS de X e Y en el orden correcto hacia adelante. La llamada inicial es PRINT-LCS.b; X; X:longitud; Y:longitud/.

```

IMPRIMIR-LCS.b; X; i; j /
1 si i == 0 o j == 0 2
volver
3 si bEi; j == “-”
4      IMPRIMIR-LCS.b; X;    1; 1/
i 5 print xi 6 elseif
bEi; j == “”
      IMPRIMIR-LCS.b; X; yo 1; j /
7 8 más IMPRIMIR-LCS.b; X; i; 1/

```

Para la tabla b de la figura 15.8, este procedimiento imprime BCBA. El procedimiento toma un tiempo  $O(n^2)$ , ya que decrementa al menos uno de i y j en cada llamada recursiva.

### Mejorando el código

Una vez que haya desarrollado un algoritmo, a menudo encontrará que puede mejorar el tiempo o el espacio que utiliza. Algunos cambios pueden simplificar el código y mejorar los factores constantes pero, por lo demás, no producen una mejora asintótica en el rendimiento.

Otros pueden generar ahorros asintóticos sustanciales en tiempo y espacio.

En el algoritmo LCS, por ejemplo, podemos eliminar la tabla b por completo. Cada  $c[i][j]$  depende sólo de otras tres entradas de la tabla c:  $c[1][j], c[1][j-1]$  y  $c[i][j-1]$ . Dado el valor de  $c[i][j]$ , podemos determinar en tiempo  $O(1)$  cuál de estos tres valores se utilizó para calcular  $c[i][j]$ , sin mesa de inspección b. Por lo tanto, podemos reconstruir un LCS en  $O(mn)$  tiempo usando un procedimiento similar a PRINT-LCS.

(El ejercicio 15.4-2 le pide que dé el pseudocódigo.) Aunque ahorramos  $O(mn)$  de espacio con este método, el requisito de espacio auxiliar para calcular un LCS no disminuye asintóticamente, ya que necesitamos  $O(mn)$  de espacio para la tabla c de todos modos.

Sin embargo, podemos reducir los requisitos de espacio asintótico para LCS-LENGTH, ya que solo necesita dos filas de la tabla c a la vez: la fila que se está calculando y la fila anterior. (De hecho, como le pide que muestre el Ejercicio 15.4-4, podemos usar solo un poco más que el espacio para una fila de c para calcular la longitud de un LCS). Esta mejora funciona si solo necesitamos la longitud de un LCS; si necesitamos reconstruir los elementos de un LCS, la tabla más pequeña no guarda suficiente información para volver sobre nuestros pasos en el tiempo  $O(mn)$ .

### Ejercicios

#### 15.4-1

Determine un LCS de  $h = [0, 0, 1, 0, 1, 0, 1, 1]$  y  $h = [1, 0, 1, 1, 0, 1, 1, 0]$ .

#### 15.4-2

Proporcione un pseudocódigo para reconstruir un LCS a partir de la tabla c completa y las secuencias originales  $x = [x_1, x_2, \dots, x_m]$  y  $y = [y_1, y_2, \dots, y_n]$  en tiempo  $O(mn)$ , sin utilizar la tabla b.

#### 15.4-3

Proporcione una versión memorizada de LCS-LENGTH que se ejecute en  $O(mn)$  tiempo.

#### 15.4-4

Muestre cómo calcular la longitud de un LCS usando solo  $2\min(m, n)$  entradas en la tabla c más  $O(1)$  espacio adicional. Luego muestre cómo hacer lo mismo, pero usando  $\min(m, n)$  entradas más  $O(1)$  espacio adicional.

## 15.4-5

Proporcione un algoritmo  $O.n^2/$ -tiempo para encontrar la subsecuencia monótonamente creciente más larga de una secuencia de  $n$  números.

## 15.4-6 ?

Proporcione un algoritmo  $O(n \lg n)$ -time para encontrar la subsecuencia monótonamente creciente más larga de una secuencia de  $n$  números. (Sugerencia: observe que el último elemento de una subsecuencia candidata de longitud  $i$  es al menos tan grande como el último elemento de una subsecuencia candidata de longitud  $i - 1$ . Mantenga las subsecuencias candidatas vinculándolas a través de la secuencia de entrada).

## 15.5 Árboles de búsqueda binarios óptimos

Supongamos que estamos diseñando un programa para traducir texto del inglés al francés.

Para cada aparición de cada palabra en inglés en el texto, necesitamos buscar su equivalente en francés. Podríamos realizar estas operaciones de búsqueda construyendo un árbol de búsqueda binaria con  $n$  palabras en inglés como claves y sus equivalentes en francés como datos satelitales. Debido a que buscaremos en el árbol cada palabra individual del texto, queremos que el tiempo total dedicado a la búsqueda sea lo más bajo posible. Podríamos asegurar un tiempo de búsqueda  $O(\lg n)$  por ocurrencia usando un árbol rojo-negro o cualquier otro árbol de búsqueda binario balanceado. Sin embargo, las palabras aparecen con diferentes frecuencias, y una palabra de uso frecuente como *the* puede aparecer lejos de la raíz, mientras que una palabra de uso poco frecuente, como *machicolation*, aparece cerca de la raíz. Tal organización ralentizaría la traducción, ya que el número de nodos visitados al buscar una clave en un árbol de búsqueda binario es igual a uno más la profundidad del nodo que contiene la clave. Queremos que las palabras que aparecen con frecuencia en el texto se coloquen más cerca de la raíz.<sup>6</sup> Además, es posible que algunas palabras del texto no tengan traducción al francés,<sup>7</sup> y tales palabras no aparecerán en absoluto en el árbol de búsqueda binaria. ¿Cómo organizamos un árbol de búsqueda binario para minimizar el número de nodos visitados en todas las búsquedas, dado que sabemos con qué frecuencia aparece cada palabra?

Lo que necesitamos se conoce como un árbol de búsqueda binario óptimo. Formalmente, se nos da una secuencia  $KD$   $k_1; k_2; \dots; k_n$  de  $n$  claves distintas en orden ordenado (de modo que  $k_1 < k_2 < \dots < k_n$ ), y deseamos construir un árbol de búsqueda binaria a partir de estas claves.

Para cada clave  $k_i$ , tenemos una probabilidad  $p_i$  de que una búsqueda sea para  $k_i$ . Algunas búsquedas pueden ser para valores que no están en  $K$ , por lo que también tenemos  $n - C$  "claves ficticias"

<sup>6</sup>Si el tema del texto es la arquitectura de un castillo, tal vez queramos que aparezcan matacanes cerca de la raíz.

<sup>7</sup>Sí, el matacán tiene un equivalente francés: *mâchicoulis*.

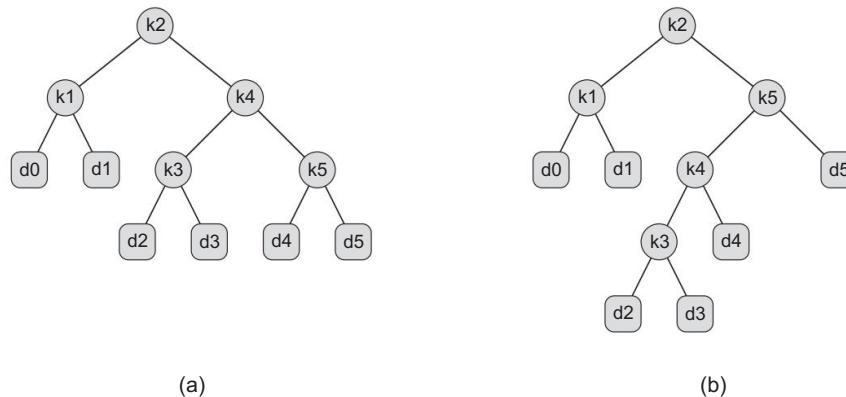


Figura 15.9 Dos árboles binarios de búsqueda para un conjunto de  $n = 5$  claves con las siguientes probabilidades:

yo	012345
pi	0,15 0,10 0,05 0,10 0,20
qi	0,05 0,10 0,05 0,05 0,05 0,10

(a) Un árbol de búsqueda binaria con un costo de búsqueda esperado de 2.80. (b) Un árbol de búsqueda binaria con un costo de búsqueda esperado de 2.75. Este árbol es óptimo.

d0; d1; d2;:::dn representa valores que no están en K. En particular, d0 representa todos los valores menores que k1, dn representa todos los valores mayores que kn, y para i D 1; 2; ::::; n1, la clave ficticia di representa todos los valores entre ki y kiC1. Para cada clave ficticia di, tenemos una probabilidad qi de que una búsqueda corresponda a di. La figura 15.9 muestra dos árboles de búsqueda binarios para un conjunto de n D 5 claves. Cada clave ki es un nodo interno y cada clave ficticia di es una hoja. Cada búsqueda es exitosa (encontrando alguna clave ki ) o fallida (encontrando alguna clave ficticia di), y así tenemos

$$X_n \pi_i C X_n \quad q D 1 \quad \dots \quad (15.10)$$

Debido a que tenemos probabilidades de búsquedas para cada clave y cada clave ficticia, podemos determinar el costo esperado de una búsqueda en un árbol de búsqueda binario T . Supongamos que el costo real de una búsqueda es igual al número de nodos examinados, es decir, la profundidad del nodo encontrado por la búsqueda en T , más 1. Entonces, el costo esperado de una búsqueda en T es

E C costo de búsqueda en TD Xn .profundidadT .ki/ C 1/ pi CXn .profundidadT .di/ C 1/ qj  
iD1 iD0

$$D_1 C_{Xn} \text{ profundidad } T .ki / pi C_{Xn} \quad \text{profundidad } T .di / qi ; \quad (15.11)$$

donde  $\text{depth}_T$  denota la profundidad de un nodo en el árbol  $T$ . La última igualdad se sigue de la ecuación (15.10). En la Figura 15.9(a), podemos calcular el costo de búsqueda esperado nodo por nodo:

nodo	profundidad	probabilidad	contribución k1	k1
0.15	k2	k3	k4	k5
d0	d1	d2		0.30
d3	0	0,10	0,10	
d4	2	0,05	0,15	
d5	1	0,10	0,20	
	2	0,20	0,60	
	2	0,05	0,15	
	2	0,10	0,30	
	3	0,05	0,20	
	3	0,05	0,20	
	3	0,05	0,20	
	3	0,10	0,40	
Total				2.80

Para un conjunto dado de probabilidades, deseamos construir un árbol de búsqueda binario cuyo costo de búsqueda esperado sea el más pequeño. A este árbol lo llamamos árbol de búsqueda binario óptimo.

La figura 15.9(b) muestra un árbol de búsqueda binario óptimo para las probabilidades dadas en el pie de figura; su costo esperado es 2.75. Este ejemplo muestra que un árbol de búsqueda binario óptimo no es necesariamente un árbol cuya altura total sea la más pequeña. Tampoco podemos construir necesariamente un árbol de búsqueda binario óptimo colocando siempre la clave con la mayor probabilidad en la raíz. Aquí, la clave k5 tiene la mayor probabilidad de búsqueda de cualquier clave, pero la raíz del árbol de búsqueda binario óptimo que se muestra es k2.

(El costo esperado más bajo de cualquier árbol de búsqueda binaria con k5 en la raíz es 2.85).

Al igual que con la multiplicación de cadenas de matrices, la verificación exhaustiva de todas las posibilidades no produce un algoritmo eficiente. Podemos etiquetar los nodos de cualquier árbol binario de  $n$  nodos con las claves  $k_1; k_2; \dots; k_n$  para construir un árbol de búsqueda binaria y luego agregar las claves ficticias como hojas. En el problema 12-4 vimos que el número de árboles binarios con  $n$  nodos es  $.4n=n^3=2^n$ , por lo que tendríamos que examinar un número exponencial de árboles binarios de búsqueda en una búsqueda exhaustiva. No en vano, resolveremos este problema con programación dinámica.

#### Paso 1: La estructura de un árbol de búsqueda binario óptimo

Para caracterizar la subestructura óptima de los árboles de búsqueda binarios óptimos, comenzamos con una observación sobre los subárboles. Considere cualquier subárbol de un árbol de búsqueda binaria.

Debe contener claves en un rango contiguo  $k_i; \dots; k_j$ , para algún  $1 \leq i, j \leq n$ .

Además, un subárbol que contiene claves  $k_i; \dots; k_j$  también debe tener como hojas las claves ficticias  $d_1; \dots; d_j$ .

Ahora podemos establecer la subestructura óptima: si un árbol de búsqueda binario óptimo  $T$  debe ser tiene un subárbol  $T'$  que contiene claves  $k_i; \dots; k_j$ , entonces este subárbol  $T'$  óptimo como

bueno para el subproblema con las teclas  $ki ;:::;kj$  y las teclas ficticias  $di1;:::;dj$ . Se aplica el argumento habitual de cortar y pegar. Si hubiera un subárbol  $T$  cuyo costo esperado fuera menor que el de  $T$ , entonces podríamos quitar  $T$  de  $T'$  y pegarlo en  $T_{00}$ , resultando en un árbol de búsqueda binaria de menor costo esperado que , contradiciendo así  $T$  la optimización de  $T$  .

Necesitamos usar la subestructura óptima para mostrar que podemos construir una solución óptima al problema a partir de soluciones óptimas a subproblemas. Dadas las claves  $ki ;:::;kj$  , una de estas claves, digamos  $kr$  ( $ij$  ), es la raíz de un subárbol óptimo que contiene estas claves. El subárbol izquierdo de la raíz  $kr$  contiene las claves  $ki ;:::;kr1$  (y las claves ficticias  $di1;:::;dr1$ ), y el subárbol derecho contiene las claves  $krC1;:::;kj$  (y las claves ficticias  $dr ; ::::;dj$  ). Siempre que examinemos todas las raíces candidatas  $kr$  , donde  $i$  y determinemos todos los árboles de búsqueda binarios óptimos  $j$  , que contengan  $ki ;:::;kr1$  y aquellos que contengan  $krC1;:::;kj$  , tenemos la garantía de que encontraremos un árbol de búsqueda binario óptimo.

Hay un detalle que vale la pena señalar sobre los subárboles "vacíos". Supongamos que en un subárbol con claves  $ki ;:::;kj$  , seleccionamos  $ki$  como raíz. Por el argumento anterior, el subárbol izquierdo de  $ki$  contiene las claves  $ki ;:::;ki1$ . Interpretamos esta secuencia como que no contiene claves. Sin embargo, tenga en cuenta que los subárboles también contienen claves ficticias. Adoptamos la convención de que un subárbol que contiene claves  $ki ;:::;ki1$  no tiene claves reales pero contiene la única clave ficticia  $di1$ . Simétricamente, si seleccionamos  $kj$  como la raíz, entonces el subárbol derecho de  $kj$  contiene las claves  $kjC1;:::;kj$  ; este subárbol derecho no contiene claves reales, pero sí contiene la clave ficticia  $dj$  .

### Paso 2: una solución recursiva

Estamos listos para definir recursivamente el valor de una solución óptima. Elegimos nuestro dominio de subproblemas para encontrar un árbol de búsqueda binario óptimo que contenga las claves  $ki ;:::;kj$  ,  $n$  y  $j$  1. (Cuando  $j \neq 1$  , no hay claves reales; solo tenemos la clave ficticia  $di1$  . ) Definamos  $e\mathbb{E}_i$ ;  $j$  como el costo esperado de buscar un árbol de búsqueda binario óptimo que contenga las claves  $ki ;:::;kj$  . En última instancia, deseamos calcular  $e\mathbb{E}_1$ ; norte.

El caso fácil ocurre cuando  $j \neq 1$  . Entonces solo tenemos la tecla ficticia  $di1$  . El costo de búsqueda esperado es  $q_1$ .

$e\mathbb{E}_i$ ;  $j$  Cuando  $j=1$  , necesitamos seleccionar una raíz  $kr$  de entre  $ki ;:::;kj$  y luego hacer un árbol de búsqueda binario óptimo con claves  $ki ;:::;kr1$  como su subárbol izquierdo y un árbol de búsqueda binario óptimo con claves  $krC1 ;:::;kj$  como su subárbol derecho. ¿Qué sucede con el costo de búsqueda esperado de un subárbol cuando se convierte en un subárbol de un nodo? La profundidad de cada nodo en el subárbol aumenta en 1. Por la ecuación (15.11), el costo de búsqueda esperado de este subárbol aumenta en la suma de todas las probabilidades en el subárbol. Para un subárbol con claves  $ki ;:::;kj$  , denotemos esta suma de probabilidades como

$$\begin{array}{ccc} j & & j \\ \text{Wisconsin;} & j / D X \text{ por } C X & q l : \\ \text{IDi} & & \text{IDi1} \end{array} \quad (15.12)$$

Por lo tanto, si  $k_r$  es la raíz de un subárbol óptimo que contiene claves  $k_i ; ; ; k_j$ , tenemos  $eOEi; j D pr C .eOEi; r1Cwi; r 1// C .eOEer C 1; j C wr C 1; j // :$

Señalando que

Wisconsin;  $j / D wi; r 1 / C pr C wr C 1; j /;$

reescribimos  $eOEi; j$

como  $eOEi; j D eOEi; r 1 C eOEer C 1; j C wi; j / : La$  (15.13)

ecuación recursiva (15.13) supone que sabemos qué nodo  $k_r$  usar como raíz. Elegimos la raíz que da el costo de búsqueda esperado más bajo, dándonos nuestra formulación recursiva final:

$$eOEi; j D \left( \min_{i,j} \begin{array}{c} feOEi; r 1 C eOEer C 1; j C wi; j / g \text{ si } ij : \\ \text{si } j D i \quad 1 \end{array} \right) \quad (15.14)$$

El  $eOEi$ ; Los valores de  $j$  dan los costos de búsqueda esperados en árboles de búsqueda binarios óptimos. Para ayudarnos a realizar un seguimiento de la estructura de los árboles de búsqueda binarios óptimos definimos  $\text{root}OEi; j$ ,  $i, j$ , para ser el índice  $r$  para el cual  $k_r$  es la raíz de un de búsqueda binario óptimo para 1 que contiene claves  $k_i ; ; ; k_j$ . Aunque veremos cómo calcular los valores de  $\text{root}OEi; j$ , dejamos la construcción de un árbol de búsqueda binario óptimo a partir de estos valores como Ejercicio 15.5-1.

### Paso 3: Cálculo del costo de búsqueda esperado de un árbol de búsqueda binario óptimo

En este punto, es posible que haya notado algunas similitudes entre nuestras caracterizaciones de árboles de búsqueda binarios óptimos y la multiplicación de cadenas de matrices. Para ambos dominios de problemas, nuestros subproblemas consisten en subrangos de índices contiguos. Una implementación recursiva directa de la ecuación (15.14) sería tan ineficiente como un algoritmo de multiplicación de cadenas de matrices directo y recursivo. En cambio, almacenamos el  $eOEi$ ; norte. El primer índice debe ejecutarse en  $nC1$  en lugar de  $n$  porque  $j$  valores en una tabla  $eOE1 : : nC1; 0$  para tener un subárbol que contenga solo la clave ficticia  $d_n$ , necesitamos calcular y almacenar  $eOE1 C 1; norte$ . El segundo índice debe comenzar desde 0 porque para tener un subárbol que contenga solo la clave ficticia  $d_0$ , debemos calcular y almacenar  $eOE1; 0$ . Usamos solo las entradas  $eOEi; j$  para lo cual  $ji 1$ . También usamos una raíz de tabla  $OEi; j$ , para registrar la raíz del subárbol que contiene las claves  $k_i ; ; ; k_j$ . Esta tabla utiliza sólo las entradas para las que  $1 \leq j \leq n$ .

Necesitaremos otra mesa para ser eficientes. En lugar de calcular el valor de  $wi; j$  desde cero cada vez que estamos computando  $eOEi; j$  —que tomaría

, $j_i$  / adiciones—almacenamos estos valores en una tabla  $w_{C1} \dots n C 1; 0$  caso norte. Para el base, calculamos  $w_{C1}; i 1 D qj1$  para 1 en  $C 1$ . Para  $j_i$ , calculamos

$$w_{C1}; j D w_{C1}; j 1 C pj C qj : \quad (15.15)$$

Por lo tanto, podemos calcular los valores de  $,n2/$  de  $w_{C1}; j$  en  $,1/$  vez cada uno.

El pseudocódigo que sigue toma como entradas las probabilidades  $p1;\dots;p_n$  y  $q0;\dots;q_n$  y el tamaño  $n$ , y devuelve las tablas  $e$  y  $\text{root}$ .

ÓPTIMA-BST.p;  $q$ ;  $n/ 1$  sea

```

eC1 :: :: n, wC1; y: rootC1; 0: m, C1 12 para i D 1 a n C 1
eC1; i 3 1 D qj1      n ser tablas nuevas
wC1; i 4 1 D qj1 5 para i D
6 para i D 1 a m C 1 j D i C 17
j D 1 9 wC1; eC1; wC1; j 1 C pj C
qj 10 para r D i a j 11
t D eC1; r 1 C eC1; C 1; j C wC1; j 12
si t < eC1; j      1
13 eC1; j D t 14 rootC1; j Dr 15
devuelve e y raíz

```

A partir de la descripción anterior y la similitud con el procedimiento MATRIX-CHAIN-ORDER de la sección 15.2, encontrará que la operación de este procedimiento es bastante sencilla. El bucle for de las líneas 2 a 4 inicializa los valores de  $e_{C1}; i 1$  y  $w_{C1}; i 1$ . El ciclo for de las líneas 5–14 luego usa las recurrencias (15.14) y (15.15) para calcular  $e_{C1}; j$  y  $w_{C1}; j$  para todo  $1 \leq j \leq n$ . En la primera iteración, cuando  $i = 1$ , el bucle calcula  $e_{C1}; y$  o  $w_{C1}; i$  por  $i = 1; 2; \dots; n$ . La segunda iteración, con  $i = 2$ , calcula  $e_{C1}; iC1$  y  $w_{C1}; iC1$  para  $i = 1; 2; \dots; n-1$ , y así sucesivamente. El bucle for más interno, en las líneas 10 a 14, prueba cada índice candidato  $r$  para determinar qué clave  $k_r$  usar como la raíz de un árbol de búsqueda binario óptimo que contiene las claves  $k_1; \dots; k_j$ .

Este bucle for guarda el valor actual del índice  $r$  en  $\text{root}_{C1}; j$  siempre que encuentre una clave mejor para usar como raíz.

La Figura 15.10 muestra las tablas  $e_{C1}; j$ ,  $w_{C1}; j$  y  $\text{root}_{C1}; j$  calculado por el procedimiento OPTIMAL-BST en la distribución de claves que se muestra en la Figura 15.9. Como en el ejemplo de multiplicación en cadena de matrices de la figura 15.5, las tablas se giran para hacer

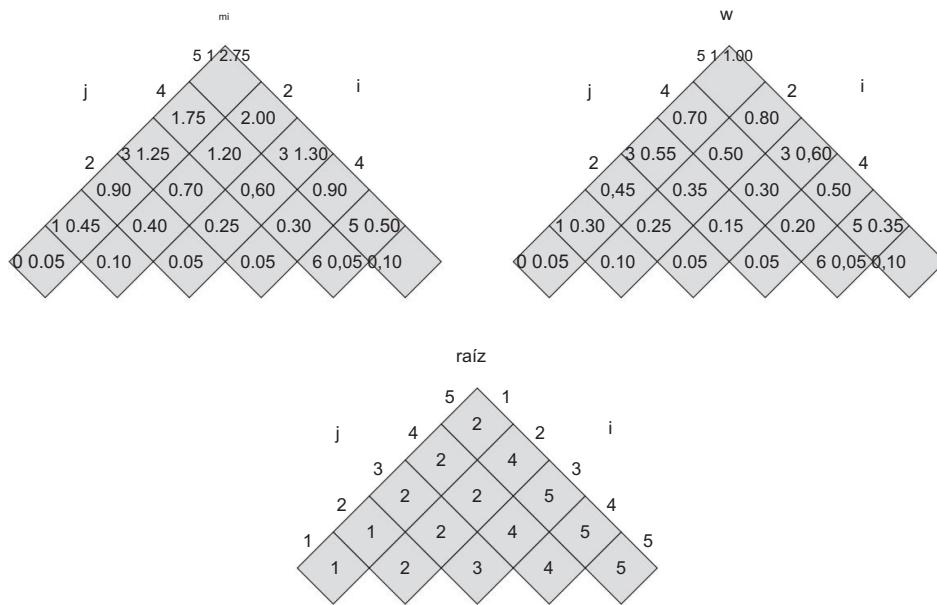


Figura 15.10 Las tablas e*C<sub>i</sub>*; j, w*C<sub>i</sub>*; j y root*C<sub>i</sub>*; j calculado por OPTIMAL-BST en la distribución de claves que se muestra en la Figura 15.9. Las mesas se giran para que las diagonales corran horizontalmente.

las diagonales corren horizontalmente. OPTIMAL-BST calcula las filas de abajo hacia arriba y de izquierda a derecha dentro de cada fila.

El procedimiento OPTIMAL-BST toma  $.n^3$ / tiempo, al igual que MATRIX-CHAIN ORDER. Podemos ver fácilmente que su tiempo de ejecución es  $O.n^3$ , ya que sus bucles for están anidados en tres profundidades y cada índice de bucle toma como máximo n valores. Los índices de bucle en OPTIMAL-BST no tienen exactamente los mismos límites que los de MATRIX-CHAIN ORDER, pero están dentro de 1 como máximo en todas las direcciones. Así, como el ORDEN MATRIZ-CADENA , el procedimiento OPTIMAL-BST toma  $.n^3$ / tiempo.

## Ejercicios

15.5-1

Escriba un pseudocódigo para el procedimiento CONSTRUCT-OPTIMAL-BST.root/ que, dada la raíz de la tabla, genera la estructura de un árbol de búsqueda binario óptimo. Para el ejemplo de la figura 15.10, su procedimiento debe imprimir la estructura

k2 es la raíz k1  
 es el hijo izquierdo de k2  
 d0 es el hijo izquierdo de  
 k1 d1 es el hijo derecho de  
 k1 k5 es el hijo derecho de  
 k2 k4 es el hijo izquierdo  
 de k5 k3 es el hijo izquierdo  
 de k4 d2 es el hijo izquierdo  
 hijo de k3 d3 es el hijo  
 derecho de k3 d4 es el hijo  
 derecho de k4 d5 es el hijo derecho de k5

correspondiente al árbol de búsqueda binario óptimo que se muestra en la figura 15.9(b).

#### 15.5-2

Determine el costo y la estructura de un árbol de búsqueda binario óptimo para un conjunto de  $n=7$  claves con las siguientes probabilidades:

yo	b1234567	
		0,04 0,06 0,08 0,02 0,10 0,12 0,14 pi qj 0,06
0,06	0,06 0,06 0,05 0,05 0,05 0,05 0,05	

#### 15.5-3

Suponga que en lugar de mantener la tabla  $w[i;j]$ , calculamos el valor de  $w[i;j]$  directamente de la ecuación (15.12) en la línea 9 de OPTIMAL-BST y utilizó este valor calculado en la línea 11. ¿Cómo afectaría este cambio al tiempo de ejecución asintótico de OPTIMAL-BST?

#### 15.5-4 ?

Knuth [212] ha demostrado que siempre hay raíces de subárboles óptimos tales que  $\text{root}[i;j] = \text{root}[i;j-1]$  para todo  $1 < j < n$ . Utilice este hecho para modificar el procedimiento OPTIMAL-BST para que se ejecute en el tiempo  $O(n^2)$ .

## Problemas

### 15-1 Camino simple más largo en un grafo acíclico dirigido

Supongamos que tenemos un grafo acíclico dirigido  $G = (V, E)$  con pesos de borde de valor real y dos vértices distinguidos  $s$  y  $t$ . Describir un enfoque de programación dinámica para encontrar la ruta simple ponderada más larga de  $s$  a  $t$ .

¿Cómo se ve el gráfico del subproblema? ¿Cuál es la eficiencia de su algoritmo?

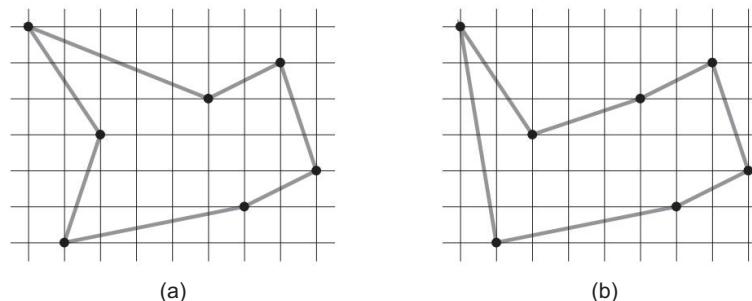


Figura 15.11 Siete puntos en el plano, mostrados en una cuadrícula unitaria. (a) El recorrido cerrado más corto, con una duración aproximada de 24:89. Este recorrido no es bitónico. (b) El recorrido bitónico más corto para el mismo conjunto de puntos. Su duración es de aproximadamente 25:58.

### 15-2 Subsecuencia palíndromo más larga Un

palíndromo es una cadena no vacía sobre algún alfabeto que se lee igual hacia adelante y hacia atrás. Ejemplos de palíndromos son todas las cadenas de longitud 1, cívica, de carreras y aibofobia (miedo a los palíndromos).

Proporcione un algoritmo eficiente para encontrar el palíndromo más largo que sea una subsecuencia de una cadena de entrada dada. Por ejemplo, dado el carácter de entrada, su algoritmo debería devolver carac. ¿Cuál es el tiempo de ejecución de su algoritmo?

15-3 Problema del viajante de comercio euclíadiano bitónico En el problema del viajante de comercio euclíadiano, tenemos un conjunto de  $n$  puntos en el plano, y deseamos encontrar el recorrido cerrado más corto que conecte todos los  $n$  puntos.

La figura 15.11(a) muestra la solución a un problema de 7 puntos. El problema general es NP-difícil y, por lo tanto, se cree que su solución requiere más tiempo que un polinomio (véase el capítulo 34).

JL Bentley ha sugerido que simplifiquemos el problema restringiendo nuestra atención a los recorridos bitónicos, es decir, recorridos que comienzan en el punto más a la izquierda, van estrictamente hacia la derecha hasta el punto más a la derecha y luego van estrictamente hacia la izquierda de regreso al punto de partida. La Figura 15.11(b) muestra el recorrido bitónico más corto de los mismos 7 puntos. En este caso, es posible un algoritmo de tiempo polinomial.

Describir un algoritmo  $O.n^2/\text{tiempo}$  para determinar un recorrido bitónico óptimo. Puede suponer que no hay dos puntos que tengan la misma coordenada  $x$  y que todas las operaciones con números reales toman la unidad de tiempo. (Sugerencia: escanee de izquierda a derecha, manteniendo las posibilidades óptimas para las dos partes del recorrido).

### 15-4 Imprimir con claridad

Considere el problema de imprimir con claridad un párrafo con una fuente monoespaciada (todos los caracteres tienen el mismo ancho) en una impresora. El texto de entrada es una secuencia de  $n$

palabras de longitud  $l_1; l_2; \dots; l_n$ , medido en caracteres. Queremos imprimir este párrafo claramente en varias líneas que contengan un máximo de  $M$  caracteres cada una. Nuestro criterio de "pulcritud" es el siguiente. Si una línea dada contiene palabras  $i$  a  $j$ , donde  $i, j$ , y dejamos exactamente un espacio entre palabras, el número de espacios adicionales al final de la línea es  $M - \sum_{k=i}^j l_k$ , que debe ser no negativo para que las palabras caben en la línea. Deseamos minimizar la suma, en todas las líneas excepto la última, de los cubos de los números de caracteres de espacio extra al final de las líneas. Proporcione un algoritmo de programación dinámica para imprimir un párrafo de  $n$  palabras de forma ordenada en una impresora. Analice los requisitos de espacio y tiempo de ejecución de su algoritmo.

### 15-5 Editar distancia

Para transformar una cadena de texto de origen  $x[1:m]$  en una cadena de destino  $y[1:n]$ , podemos realizar varias operaciones de transformación. Nuestro objetivo es, dados  $x$  e  $y$ , producir una serie de transformaciones que cambien  $x$  a  $y$ . Usamos una matriz  $\emptyset$ , que se supone que es lo suficientemente grande como para contener todos los caracteres que necesitará, para contener los resultados intermedios. Inicialmente,  $\emptyset$  está vacío, y al terminar, deberíamos tener  $\emptyset[j:D-1] = y[D-1:n]$ . Mantenemos los índices actuales  $i$  en  $x$  y  $j$  en  $\emptyset$ , y las operaciones pueden alterar  $\emptyset$  y estos índices. Inicialmente,  $i=D-1$  y  $j=D-1$ . Estamos obligados a examinar cada carácter en  $x$  durante la transformación, lo que significa que al final de la secuencia de operaciones de transformación, debemos tener  $i=m$  y  $j=1$ .

Podemos elegir entre seis operaciones de transformación:

Copie un carácter de  $x$  a  $\emptyset$  configurando  $\emptyset[j:D-1] = x[i]$  y luego incrementando ambos  $i$ . Esta operación examina  $x[i:j]$ .

Reemplace un carácter de  $x$  por otro carácter  $c$ , configurando  $\emptyset[j:D-1] = c$ , y luego Esta operación examina  $x[i:j]$ .

Elimine un carácter de  $x$  incrementando  $i$  pero dejando  $j$  solo. Esta operación examina  $x[i:j]$ .

Inserte el carácter  $c$  en  $\emptyset$  configurando  $\emptyset[j:D-1] = c$  y luego incrementando  $j$ , pero dejando  $i$  solo. Esta operación no examina caracteres de  $x$ .

Haga girar (es decir, intercambie) los siguientes dos caracteres copiándolos de  $x$  a  $\emptyset$  pero en el orden opuesto; lo hacemos al establecer  $\emptyset[j:D-1] = x[i:D-1]$  y  $\emptyset[i:D-1] = x[D:j]$  y luego establecer  $i=D-1$  y  $j=D-1$ . Esta operación examina  $x[i:j]$  y  $x[D:j]$ .

Elimine el resto de  $x$  configurando  $i=m$  y  $j=1$ . Esta operación examina todos los caracteres en  $x$  que aún no han sido examinados. Esta operación, si se realiza, debe ser la operación final.

Como ejemplo, una forma de transformar el algoritmo de cadena de origen en el altruista de cadena de destino es usar la siguiente secuencia de operaciones, donde los caracteres subrayados son  $\text{x} \rightarrow \text{y}$  después de la operación:

Operación	X	
<u>cadenas iniciales algoritmo</u>		
Copiar	<u>algoritmo</u>	a _
copiar	<u>algoritmo</u>	Alabama _
reemplazar por	<u>algoritmo</u>	alternativa _
t borrar	<u>algoritmo</u>	alternativa _
copiar	<u>algoritmo</u>	altr _
insertar u	<u>algoritmo</u>	altru _
insertar yo	<u>algoritmo</u>	altrui _
insertar s	<u>algoritmo</u>	altruis _
girar	<u>algoritmo</u>	altruista _
insertar c	_	altruista _
matar	_	altruista _

Tenga en cuenta que hay varias otras secuencias de operaciones de transformación que transforman el algoritmo en altruista.

Cada una de las operaciones de transformación tiene un coste asociado. El costo de una operación depende de la aplicación específica, pero asumimos que el costo de cada operación es una constante que conocemos. También suponemos que los costos individuales de las operaciones de copiar y reemplazar son menores que los costos combinados de las operaciones de borrar e insertar; de lo contrario, no se utilizarían las operaciones de copiar y reemplazar. El costo de una secuencia dada de operaciones de transformación es la suma de los costos de las operaciones individuales en la secuencia. Para la secuencia anterior, el costo de transformar el algoritmo en altruista es

.3 costo.copiar// C costo.reemplazar/ C costo.borrar/ C .4 costo.insertar//

C costo.twiddle/ C costo.matar/ :

- Dadas dos secuencias  $x \rightarrow y$  y un conjunto de costos de operación de transformación, la distancia de edición de  $x$  a  $y$  es el costo de la secuencia de operación menos costosa que transforma  $x$  en  $y$ . Describa un algoritmo de programación dinámica que encuentre la distancia de edición desde  $x \rightarrow y$  hasta  $y \rightarrow z$  e imprima una secuencia de operación óptima. Analice los requisitos de espacio y tiempo de ejecución de su algoritmo.

El problema de la distancia de edición generaliza el problema de alinear dos secuencias de ADN (ver, por ejemplo, Setubal y Meidanis [310, Sección 3.2]). Hay varios métodos para medir la similitud de dos secuencias de ADN alineándolas.

Uno de estos métodos para alinear dos secuencias  $x$  e  $y$  consiste en insertar espacios en

ubicaciones arbitrarias en las dos secuencias (incluso en cualquier extremo) de modo que las secuencias resultantes  $x_0$  y  $y_0$  tengan la misma longitud pero no tengan un espacio en la misma posición (es decir, para ninguna posición  $j$  son  $x_0[j]$  y  $y_0[j]$  a espacio). Luego asignamos una "puntuación" a cada puesto. La posición  $j$  recibe una puntuación de la siguiente manera:

C1 si  $x_0[j] \neq y_0[j]$  y ninguno es un espacio, 1 si

$x_0[j] = y_0[j]$  y ninguno es un espacio, 2 si  $x_0[j]$

$\neq y_0[j]$  es un espacio.

La puntuación de la alineación es la suma de las puntuaciones de las posiciones individuales. Por ejemplo, dadas las secuencias  $x$  D GATCGGCAT y  $y$  D CAATGTGAATC, una alineación es

G ATCG GCAT

CAAT GTGAATC

-----+\*+-++\*

Un + debajo de una posición indica una puntuación de C1 para esa posición, un - indica una puntuación de 1 y un \* indica una puntuación de 2, por lo que esta alineación tiene una puntuación total de 6 1 2 1 4 2 D 4.

- b. Explique cómo convertir el problema de encontrar una alineación óptima en un problema de distancia de edición utilizando un subconjunto de las operaciones de transformación copiar, reemplazar, eliminar, insertar, girar y matar.

#### 15-6 Planificación de una fiesta de

empresa El profesor Stewart está asesorando al presidente de una corporación que está organizando una fiesta de empresa. La empresa tiene una estructura jerárquica; es decir, la relación de supervisor forma un árbol con raíz en el presidente. La oficina de personal ha clasificado a cada empleado con una calificación de cordialidad, que es un número real. Para que la fiesta sea divertida para todos los asistentes, el presidente no quiere que asistan ni un empleado ni su supervisor inmediato.

Al profesor Stewart se le da el árbol que describe la estructura de la corporación, usando la representación del hijo izquierdo y el hermano derecho descrita en la Sección 10.4. Cada nodo del árbol contiene, además de los punteros, el nombre de un empleado y la clasificación de convivencia de ese empleado. Describa un algoritmo para hacer una lista de invitados que maximice la suma de las calificaciones de convivencia de los invitados. Analice el tiempo de ejecución de su algoritmo.

#### 15-7 Algoritmo de Viterbi

Podemos usar programación dinámica en un grafo dirigido  $G = (V, E)$  para reconocimiento de voz. Cada arista  $(u, v) \in E$  está etiquetado con un sonido  $.u; /$  de un conjunto finito  $\Sigma$  de sonidos. El gráfico etiquetado es un modelo formal de una persona que habla

un lenguaje restringido. Cada camino en el gráfico que parte de un vértice distinguido  $O \in V$  la etiqueta corresponde a una posible secuencia de sonidos producidos por el modelo. Definimos la etiqueta de un camino dirigido como la concatenación de las etiquetas de los bordes en ese camino.

- a. Describa un algoritmo eficiente que, dado un gráfico  $G$  con etiquetas de borde con  $\text{dis}$  y una secuencia  $s$ , determine si existe un camino en  $G$  que comienza en  $O$  y tiene la secuencia  $s$  como su etiqueta. Si existe tal camino,

De lo contrario, el algoritmo debería devolver NO-SUCH-PATH. Analice el tiempo de ejecución de su algoritmo. (Sugerencia: puede encontrar útiles los conceptos del Capítulo 22).

Ahora, suponga que cada arista  $u \in E$  tiene asociada una probabilidad no negativa  $p_u$  de atravesar la arista  $u$  desde el vértice  $u$  y produciendo así el sonido correspondiente. La suma de las probabilidades de que las aristas dejen cualquier vértice es igual a 1. La probabilidad de un camino se define como el producto de las probabilidades de sus aristas. Podemos ver la probabilidad de que un camino comience en  $O$  como la probabilidad de que una "caminata aleatoria" que comience en  $O$  siga el camino especificado, donde elegimos aleatoriamente qué borde tomar dejando un vértice  $u$  de acuerdo con las probabilidades de que los bordes disponibles dejen  $u$ .

- b. Amplíe su respuesta a la parte (a) de modo que si se devuelve una ruta, es la ruta más probable que comienza en  $O$  y tiene la etiqueta  $s$ . Analice el tiempo de ejecución de su algoritmo.

#### 15-8 Compresión de imagen por talla de costura

Se nos da una imagen en color que consta de un arreglo  $mn \times 3$ :  $m$  filas y  $n$  columnas de píxeles, donde cada píxel especifica un triple de intensidades de rojo, verde y azul (RGB). Supongamos que deseamos comprimir ligeramente esta imagen. Específicamente, deseamos eliminar un píxel de cada una de las  $m$  filas, de modo que la imagen completa se vuelva un píxel más estrecha. Sin embargo, para evitar efectos visuales perturbadores, requerimos que los píxeles eliminados en dos filas adyacentes estén en la misma columna o en columnas adyacentes; los píxeles eliminados forman una "costura" desde la fila superior hasta la fila inferior donde los píxeles sucesivos de la costura son adyacentes vertical o diagonalmente.

- a. Muestre que el número de tales costuras posibles crece al menos exponencialmente en  $m$ , suponiendo que  $n > 1$ .

- b. Supongamos ahora que junto con cada píxel  $A[i][j]$ , hemos calculado un real medida de interrupción valorada  $d[i][j]$ , que indica cuán perturbador sería eliminar el píxel  $A[i][j]$ . Intuitivamente, cuanto menor es la medida de interrupción de un píxel, más similar es el píxel a sus vecinos. Supongamos además que definimos la medida de interrupción de una costura como la suma de las medidas de interrupción de sus píxeles.

Proporcione un algoritmo para encontrar una costura con la medida de interrupción más baja.  
 ¿Qué tan eficiente es su algoritmo?

#### 15-9 Rompiendo una cadena

Certo lenguaje de procesamiento de cadenas permite a un programador dividir una cadena en dos partes. Debido a que esta operación copia la cadena, cuesta  $n$  unidades de tiempo dividir una cadena de  $n$  caracteres en dos partes. Supongamos que un programador quiere dividir una cadena en muchos pedazos. El orden en que ocurren los descansos puede afectar la cantidad total de tiempo utilizado. Por ejemplo, suponga que el programador quiere dividir una cadena de 20 caracteres después de los caracteres 2, 8 y 10 (numerando los caracteres en orden ascendente desde el extremo izquierdo, comenzando desde 1). Si ella programa las pausas para que ocurran en orden de izquierda a derecha, entonces la primera pausa cuesta 20 unidades de tiempo, la segunda pausa cuesta 18 unidades de tiempo (rompiendo la cadena de los caracteres 3 a 20 en el carácter 8) y la tercera pausa cuesta 12 unidades de tiempo, totalizando 50 unidades de tiempo. Sin embargo, si ella programa los descansos para que ocurran en orden de derecha a izquierda, entonces el primer descanso cuesta 20 unidades de tiempo, el segundo descanso cuesta 10 unidades de tiempo y el tercero cuesta 8 unidades de tiempo, totalizando 38 unidades de tiempo. En otro orden más, podría romper primero en 8 (que cuesta 20), luego romper la pieza izquierda en 2 (que cuesta 8) y finalmente la pieza derecha en 10 (que cuesta 12), por un costo total de 40.

Diseñe un algoritmo que, dada la cantidad de caracteres después de los cuales romper, determine una forma de menor costo para secuenciar esos descansos. Más formalmente, dada una cadena  $S$  con  $n$  caracteres y una matriz  $L \in \mathbb{R}^{n \times m}$  que contiene los puntos de ruptura, calcule el costo más bajo para una secuencia de rupturas, junto con una secuencia de rupturas que logre este costo.

#### 15-10 Planificación de una estrategia de inversión

Su conocimiento de los algoritmos lo ayuda a obtener un trabajo emocionante en Acme Computer Company, junto con un bono de firma de \$10,000. Decide invertir este dinero con el objetivo de maximizar su retorno al final de 10 años. Decide utilizar Amalgamated Investment Company para gestionar sus inversiones.

Amalgamated Investments requiere que observe las siguientes reglas. Ofrece  $n$  inversiones diferentes, numeradas del 1 al  $n$ . En cada año  $j$ , la inversión  $i$  proporciona En otras palabras, si luego, al final del año  $j$  invierte  $d$  dólares en la inversión  $i$  en el año  $j$ , una tasa de retorno de  $r_{ij}$ .  $j$ , tienes dólares  $d_{rij}$ . Las tasas de retorno están garantizadas, es decir, se le dan todas las tasas de retorno de los próximos 10 años para cada inversión.

Usted toma decisiones de inversión solo una vez al año. Al final de cada año, puede dejar el dinero ganado en el año anterior en las mismas inversiones, o puede transferir dinero a otras inversiones, ya sea transfiriendo dinero entre inversiones existentes o transfiriendo dinero a una nueva inversión. Si no mueve su dinero entre dos años consecutivos, paga una tarifa de  $f_1$  dólares, mientras que si cambia su dinero, paga una tarifa de  $f_2$  dólares, donde  $f_2 > f_1$ .

- a. El problema, como se dijo, le permite invertir su dinero en múltiples inversiones cada año. Demostrar que existe una estrategia de inversión óptima que, en cada año, pone todo el dinero en una sola inversión. (Recuerde que una estrategia de inversión óptima maximiza la cantidad de dinero después de 10 años y no se preocupa por ningún otro objetivo, como minimizar el riesgo).
- b. Demuestre que el problema de planificar su estrategia de inversión óptima exhibe una subestructura óptima.
- C. Diseña un algoritmo que planifique tu estrategia de inversión óptima. ¿Cuál es el tiempo de ejecución de su algoritmo?
- d. Suponga que Amalgamated Investments impuso la restricción adicional de que, en cualquier momento, no puede tener más de \$15,000 en cualquier inversión. Demuestre que el problema de maximizar su ingreso al final de 10 años ya no presenta una subestructura óptima.

#### 15-11 Planificación de

inventario La empresa Rinky Dink fabrica máquinas que repavimentan pistas de hielo. La demanda de dichos productos varía de un mes a otro, por lo que la empresa necesita desarrollar una estrategia para planificar su fabricación dada la demanda fluctuante pero predecible. La empresa desea diseñar un plan para los próximos  $n$  meses. Para cada mes  $i$ , la empresa conoce la demanda  $D_i$ , es decir, la cantidad de máquinas que venderá. Sea  $P_n$  la demanda total durante los próximos  $n$  meses. La empresa mantiene un personal de tiempo completo que proporciona mano de obra para fabricar hasta máquinas por mes. Si la empresa necesita fabricar más de  $m$  máquinas en un mes determinado, puede contratar mano de obra adicional a tiempo parcial, a un costo de  $c$  dólares por máquina. Además, si, al final de un mes, la empresa tiene máquinas sin vender, debe pagar los costos de inventario. El costo de mantener  $j$  máquinas está dado como una función  $H_j$  para  $j = 1, 2, \dots, D$ , donde  $H_0 = 0$  para  $1 \leq j \leq D$  y  $H_j = C_j$  para  $j > D$ .

Proporcione un algoritmo que calcule un plan para la empresa que minimice sus costos y satisfaga toda la demanda. El tiempo de ejecución debe ser polomial en  $n$  y  $D$ .

#### 15-12 Contratación de jugadores de béisbol

agentes libres Suponga que usted es el gerente general de un equipo de béisbol de las ligas mayores. Durante la temporada baja, debe fichar a algunos jugadores agentes libres para su equipo. El dueño del equipo te ha dado un presupuesto de  $X$  para gastar en agentes libres. Puede gastar menos de  $X$  en total, pero el propietario lo despedirá si gasta más de  $X$ .

Está considerando N posiciones diferentes, y para cada posición, P jugadores agentes libres que juegan en esa posición están disponibles . un agente libre que juega en esa posición. (Si no firma a ningún jugador en una posición en particular, entonces planea quedarse con los jugadores que ya tiene en esa posición).

Para determinar qué tan valioso será un jugador, decide usar una estadística sabremétrica<sup>8</sup> conocida como "VORP" o "valor sobre el jugador de reemplazo". Un jugador con un VORP más alto es más valioso que un jugador con un VORP más bajo. Un jugador con un VORP más alto no es necesariamente más caro de fichar que un jugador con un VORP más bajo, porque otros factores además del valor de un jugador determinan cuánto cuesta ficharlo.

Para cada jugador agente libre disponible, tiene tres datos:

- la posición del jugador,
- la cantidad de dinero que costará fichar al jugador y el
- VORP del jugador.

Diseñe un algoritmo que maximice el VORP total de los jugadores que contrate sin gastar más de \$X en total. Puede suponer que cada jugador firma por un múltiplo de \$100,000. Su algoritmo debe generar el VORP total de los jugadores que firma, la cantidad total de dinero que gasta y una lista de los jugadores que firma. Analice el tiempo de ejecución y los requisitos de espacio de su algoritmo.

## Notas del capítulo

R. Bellman comenzó el estudio sistemático de la programación dinámica en 1955. La palabra “programación”, tanto aquí como en programación lineal, se refiere al uso de un método de solución tabular. Aunque las técnicas de optimización que incorporan elementos de programación dinámica se conocían antes, Bellman proporcionó al área una base matemática sólida [37].

<sup>8</sup>Aunque hay nueve puestos en un equipo de béisbol, N no es necesariamente igual a 9 porque algunos gerentes generales tienen formas particulares de pensar acerca de los puestos. Por ejemplo, un gerente general puede considerar que los lanzadores diestros y los zurdos son "posiciones" separadas, así como los lanzadores abridores, los lanzadores de relevo largo (lanzadores de relevo que pueden lanzar varias entradas) y los lanzadores de relevo corto (lanzadores de relevo que normalmente lanzan como máximo una sola entrada).

<sup>9</sup>Sabermetrics es la aplicación del análisis estadístico a los récords de béisbol. Proporciona varias formas de comparar los valores relativos de jugadores individuales.

Galil y Park [125] clasifican los algoritmos de programación dinámica según el tamaño de la tabla y el número de otras entradas de la tabla de las que depende cada entrada. Llaman a un algoritmo de programación dinámica  $tD=eD$  si el tamaño de su tabla es  $O.n^t$  / y cada entrada depende de  $O.n^e$  otras entradas. Por ejemplo, el algoritmo de multiplicación matriz-cadena de la Sección 15.2 sería  $2D=1D$ , y el algoritmo de la subsecuencia común más larga de la Sección 15.4 sería  $2D=0D$ .

Hu y Shing [182, 183] dan un algoritmo  $O(n \lg n)$ -time para el problema de multiplicación de matriz-cadena.

El algoritmo  $O(mn)$ -time para el problema de la subsecuencia común más larga parece ser un algoritmo popular. Knuth [70] planteó la cuestión de si existen algoritmos subcuadráticos para el problema LCS. Masek y Paterson [244] respondieron afirmativamente a esta pregunta proporcionando un algoritmo que se ejecuta en  $O(mn) = \lg n$  time, donde  $m$  y  $n$  las secuencias se extraen de un conjunto de tamaño acotado. Para el caso especial en el que ningún elemento aparece más de una vez en una secuencia de entrada, Szymanski [326] muestra cómo resolver el problema en tiempo  $O(n \lg m)$ .

Muchos de estos resultados se extienden al problema de calcular distancias de edición de cadenas (Problema 15-5).

Un artículo anterior sobre codificaciones binarias de longitud variable de Gilbert y Moore [133] tenía aplicaciones para construir árboles de búsqueda binarios óptimos para el caso en el que todas las probabilidades  $p_i$  son 0 ; este artículo contiene un algoritmo  $O(n^3)$ -time. Aho, Hopcroft y Ullman [5] presentan el algoritmo de la Sección 15.5. El ejercicio 15.5-4 se debe a Knuth [212]. Hu y Tucker [184] diseñaron un algoritmo para el caso en el que todas las probabilidades  $p_i$  son 0 que usa  $O(n^2)$  tiempo y  $O(n)$  espacio; posteriormente, Knuth [211] redujo el tiempo a  $O(n \lg n)$ .

El problema 15-8 se debe a Avidan y Shamir [27], quienes publicaron en la Web un maravilloso video que ilustra esta técnica de compresión de imágenes.

## Algoritmos codiciosos

Los algoritmos para problemas de optimización normalmente pasan por una secuencia de pasos, con un conjunto de opciones en cada paso. Para muchos problemas de optimización, usar programación dinámica para determinar las mejores opciones es una exageración; Algoritmos más simples y eficientes servirán. Un algoritmo codicioso siempre toma la decisión que se ve mejor en ese momento. Es decir, hace una elección localmente óptima con la esperanza de que esta elección conduzca a una solución globalmente óptima. Este capítulo explora los problemas de optimización para los cuales los algoritmos voraces proporcionan soluciones óptimas. Antes de leer este capítulo, debe leer acerca de la programación dinámica en el Capítulo 15, particularmente en la Sección 15.3.

Los algoritmos codiciosos no siempre brindan soluciones óptimas, pero sí lo hacen para muchos problemas. Primero examinaremos, en la Sección 16.1, un problema simple pero no trivial, el problema de selección de actividad, para el cual un algoritmo voraz calcula eficientemente una solución óptima. Llegaremos al algoritmo codicioso considerando primero un enfoque de programación dinámica y luego demostrando que siempre podemos tomar decisiones codiciosas para llegar a una solución óptima. La Sección 16.2 revisa los elementos básicos del enfoque voraz, brindando un enfoque directo para probar que los algoritmos voraz son correctos. La sección 16.3 presenta una aplicación importante de las técnicas codiciosas: el diseño de códigos de compresión de datos (Huffman). En la Sección 16.4, investigamos parte de la teoría que subyace a las estructuras combinatorias llamadas “matroides”, para las cuales un algoritmo voraz siempre produce una solución óptima. Finalmente, la Sección 16.5 aplica matroides para resolver un problema de programación de tareas de unidad de tiempo con plazos y penalizaciones.

El método codicioso es bastante poderoso y funciona bien para una amplia gama de problemas. Los capítulos posteriores presentarán muchos algoritmos que podemos ver como aplicaciones del método codicioso, incluidos los algoritmos de árbol de expansión mínima (Capítulo 23), el algoritmo de Dijkstra para las rutas más cortas desde una sola fuente (Capítulo 24) y el algoritmo codicioso de Chvatal. heuristic de cobertura de conjuntos (capítulo 35). Los algoritmos de árbol de expansión mínima proporcionan un ejemplo clásico del método codicioso. Aunque puedes leer

este capítulo y el Capítulo 23 de forma independiente, puede que le resulte útil leerlos juntos.

### 16.1 Un problema de selección de actividades

Nuestro primer ejemplo es el problema de programar varias actividades en competencia que requieren el uso exclusivo de un recurso común, con el objetivo de seleccionar un conjunto de tamaño máximo de actividades mutuamente compatibles. Supongamos que tenemos un conjunto  $S = \{a_1, a_2, \dots, a_n\}$  de  $n$  actividades propuestas que desean utilizar un recurso, como una sala de conferencias, que solo puede servir para una actividad a la vez. Cada actividad  $a_i$  tiene un tiempo de inicio  $s_i$  y un tiempo de finalización  $f_i$ , donde  $0 \leq s_i < f_i \leq 1$ . Si se selecciona, la actividad  $a_i$  tiene lugar durante el intervalo de tiempo semiabierto  $[s_i, f_i]$ . Las actividades  $a_i$  y  $a_j$  son compatibles si los intervalos  $[s_i, f_i]$  y  $[s_j, f_j]$  no se superponen. Es decir,  $a_i$  y  $a_j$  son compatibles si  $s_j > f_i$ . En el problema de selección de actividades, deseamos seleccionar un subconjunto de tamaño máximo o subconjunto de actividades mutuamente compatibles. Suponemos que las actividades están ordenadas en orden monótonamente creciente de tiempo de finalización:

$$f_1 \ f_2 \ f_3 \quad f_{n-1} \ f_n : \quad (16.1)$$

(Veremos más adelante la ventaja que proporciona esta suposición.) Por ejemplo, considere el siguiente conjunto  $S$  de actividades:

i	1 2 3 4 5 6 7	8	9 10 11
si	1 3 0 5 3 5 6	8	8 2 12
fi	4 5 6 7 9 9 10 11 12 14 16		

Para este ejemplo, el subconjunto  $\{a_3, a_9, a_{11}\}$  consiste en actividades mutuamente compatibles. Sin embargo, no es un subconjunto máximo, ya que el subconjunto  $\{a_1, a_4, a_8, a_{11}\}$  es más grande. De hecho,  $\{a_1, a_4, a_8, a_{11}\}$  es un subconjunto más grande de actividades mutuamente compatibles; otro subconjunto más grande es  $\{a_2, a_4, a_9, a_{11}\}$ .

Resolvemos este problema en varios pasos. Comenzamos pensando en una solución de programación dinámica, en la que consideramos varias opciones al determinar qué subproblemas usar en una solución óptima. Luego observaremos que necesitamos considerar solo una opción, la elección codiciosa, y que cuando hacemos la elección codiciosa, solo queda un subproblema. Con base en estas observaciones, desarrollaremos un algoritmo voraz recursivo para resolver el problema de programación de actividades. Completaremos el proceso de desarrollo de una solución codiciosa convirtiendo el algoritmo recursivo en uno iterativo. Aunque los pasos que seguiremos en esta sección son un poco más complicados de lo normal cuando se desarrolla un algoritmo voraz, ilustran la relación entre los algoritmos voraz y la programación dinámica.

### La subestructura óptima del problema de selección de actividades

Podemos verificar fácilmente que el problema de selección de actividades presenta una subestructura óptima. Denotemos por  $S_{ij}$  el conjunto de actividades que comienzan después de que termina la actividad  $a_i$  y que terminan antes de que comience la actividad  $a_j$ . Supongamos que deseamos encontrar un conjunto máximo de actividades mutuamente compatibles en  $S_{ij}$ , y supongamos además que dicho conjunto máximo es  $A_{ij}$ , que incluye alguna actividad  $a_k$ . Al incluir  $a_k$  en una solución óptima, nos quedan dos subproblemas: encontrar actividades mutuamente compatibles en el conjunto  $S_{ik}$  (actividades que comienzan después de que termina la actividad  $a_i$  y que terminan antes de que comience la actividad  $a_k$ ) y encontrar actividades mutuamente compatibles en el conjunto  $S_{kj}$  (actividades que comienzan después de que termina la actividad  $a_k$  y que terminan antes de que comience la actividad  $a_j$ ). Sean  $A_{ik} \subseteq A_{ij} \setminus S_{ik}$  y  $A_{kj} \subseteq A_{ij} \setminus S_{kj}$ , de modo que  $A_{ik}$  contiene las actividades en  $A_{ij}$  que terminan antes de que comience  $a_k$  y  $A_{kj}$  contiene las actividades en  $A_{ij}$  que comienzan después de que termina  $a_k$ . Por lo tanto, tenemos  $A_{ij} = A_{ik} \cup A_{kj}$ , por lo que el conjunto de tamaño máximo  $A_{ij}$  de actividades mutuamente compatibles en  $S_{ij}$  consta de  $A_{ik} \cup A_{kj}$ .

El argumento habitual de cortar y pegar muestra que la solución óptima  $A_{ij}$  también debe ser  $S_{ij}$  incluir soluciones óptimas a los dos subproblemas para  $S_{ik}$  y  $S_{kj}$ . Para encontrar  $A_{ik}$  y  $A_{kj}$ , supiéramos que un conjunto  $A_0$  de actividades mutuamente compatibles en  $S_{kj}$  donde  $j \in A_0$  en  $k > j$ , entonces  $k \in A_0$  y  $j \in A_0$ . Podríamos usar  $A_0$  en lugar de  $A_{kj}$ , en una solución al subproblema para  $S_{ik}$ . Habríamos construido un conjunto de  $A_{ik} \cup A_0 \cup A_{kj}$  en  $C_{ik} \cup C_{kj}$  actividades mutuamente compatibles, lo que contradice la suposición de que  $A_{ik}$  es una solución óptima. Un argumento simétrico se aplica a las actividades en  $S_{ik}$ .

Esta forma de caracterizar la subestructura óptima sugiere que podríamos resolver el problema de selección de actividades mediante programación dinámica. Si denotamos el tamaño de una solución óptima para el conjunto  $S_{ij}$  por  $c_{ij}$ , entonces tendríamos la recursión

$$c_{ij} = \max_{k \in S_{ij}} c_{ik} + c_{kj}$$

Por supuesto, si no supiéramos que una solución óptima para el conjunto  $S_{ij}$  incluye la actividad  $a_k$ , tendríamos que examinar todas las actividades en  $S_{ij}$  para encontrar cuál elegir, de modo que

$$c_{ij} = \begin{cases} \max_{k \in S_{ij}} c_{ik} + c_{kj} & \text{si } S_{ij} \neq \emptyset \\ 0 & \text{si } S_{ij} = \emptyset \end{cases} \quad (16.2)$$

Entonces podríamos desarrollar un algoritmo recursivo y memorizarlo, o podríamos trabajar de abajo hacia arriba y completar las entradas de la tabla a medida que avanzamos. Pero estaríamos pasando por alto otra característica importante del problema de selección de actividades que podemos utilizar con gran ventaja.

### Haciendo la elección codicosa

¿Qué pasaría si pudiéramos elegir una actividad para agregar a nuestra solución óptima sin tener que resolver primero todos los subproblemas? Eso podría ahorrarnos tener que considerar todas las opciones inherentes a la recurrencia (16.2). De hecho, para el problema de selección de actividades, necesitamos considerar solo una opción: la elección codicosa.

¿Qué queremos decir con la elección codicosa para el problema de selección de actividades? La intuición sugiere que deberíamos elegir una actividad que deje el recurso disponible para tantas otras actividades como sea posible. Ahora, de las actividades que terminemos eligiendo, una de ellas debe ser la primera en terminar. Nuestra intuición nos dice, por lo tanto, que elijamos la actividad en  $S$  con el tiempo de finalización más temprano, ya que eso dejaría el recurso disponible para tantas de las actividades que le siguen como sea posible. (Si más de una actividad en  $S$  tiene la hora de finalización más temprana, entonces podemos elegir cualquiera de esas actividades). En otras palabras, dado que las actividades se clasifican en orden monótonamente creciente según la hora de finalización, la elección codicosa es la actividad  $a_1$ . Elegir la primera actividad para terminar no es la única forma de pensar en hacer una elección codicosa para este problema; El ejercicio 16.1-3 le pide que explore otras posibilidades.

Si tomamos la decisión codicosa, solo nos queda un subproblema por resolver: encontrar actividades que comiencen después de que termine  $a_1$ . ¿Por qué no tenemos que considerar las actividades que terminan antes de que comience  $a_1$ ? Tenemos que  $s_1 < f_1$ , y  $f_1$  es el tiempo de finalización más temprano de cualquier actividad  $y$ , por lo tanto, ninguna actividad puede tener un tiempo de finalización menor o igual que  $s_1$ . Por lo tanto, todas las actividades que son compatibles con la actividad  $a_1$  deben comenzar después de que termine  $a_1$ .

Además, ya hemos establecido que el problema de selección de actividades exhibe una subestructura óptima. Sea  $S_k$  el conjunto de actividades que comienzan después de que finaliza la actividad  $a_k$ . Si hacemos la elección codicosa de la actividad  $a_1$ , entonces  $S_1$  permanece como el único subproblema a resolver.<sup>1</sup> La subestructura óptima nos dice que si  $a_1$  está en la solución óptima, entonces una solución óptima al problema original consiste en la actividad  $a_1$  y todas las actividades en una solución óptima al subproblema  $S_1$ .

Queda una gran pregunta: ¿es correcta nuestra intuición? ¿La elección codicosa, en la que elegimos la primera actividad para terminar, es siempre parte de alguna solución óptima? El siguiente teorema demuestra que lo es.

---

<sup>1</sup>A veces nos referimos a los conjuntos  $S_k$  como subproblemas en lugar de simples conjuntos de actividades. Siempre quedará claro por el contexto si nos referimos a  $S_k$  como un conjunto de actividades o como un subproblema cuya entrada es ese conjunto.

**Teorema 16.1**

Considere cualquier subproblema no vacío  $S_k$  y sea  $a_m$  una actividad en  $S_k$  con el tiempo de terminación más temprano. Entonces  $a_m$  se incluye en algún subconjunto de tamaño máximo de actividades mutuamente compatibles de  $S_k$ .

**Demostración** Sea  $A_k$  un subconjunto de tamaño máximo de actividades mutuamente compatibles en  $S_k$ , y sea  $a_j$  la actividad en  $A_k$  con el tiempo de finalización más temprano. Si  $a_j \in D_m$ , hemos terminado, ya que hemos demostrado que  $a_m$  está en algún subconjunto de tamaño máximo de actividades mutuamente compatibles de  $S_k$ . Si  $a_j \neq a_m$ , sea  $A_0 = D_m \setminus A_k$  y  $f_{A_0} = f_m - t_{a_j}$  pero sustituyendo  $a_m$  por  $a_j$ . Las actividades en  $A_0$  son disjuntas, lo que sigue porque las actividades en  $A_k$  son disjuntas,  $a_j$  es la primera actividad en  $A_k$  en terminar, y  $f_m > f_j$ .

Como  $f_{A_0} > f_m - t_{a_j}$ , concluimos que  $A_0$  es un subconjunto de tamaño máximo de actividades mutuamente compatibles de  $S_k$ , e incluye  $a_m$ . ■

Por lo tanto, vemos que aunque podríamos resolver el problema de selección de actividades con programación dinámica, no es necesario. (Además, aún no hemos examinado si el problema de selección de actividades tiene subproblemas superpuestos). En cambio, podemos elegir repetidamente la actividad que finaliza primero, mantener solo las actividades compatibles con esta actividad y repetir hasta que no queden actividades.

Además, debido a que siempre elegimos la actividad con el tiempo de finalización más temprano, los tiempos de finalización de las actividades que elegimos deben aumentar estrictamente. Podemos considerar cada actividad solo una vez en general, en orden monótonamente creciente de tiempos de finalización.

Un algoritmo para resolver el problema de selección de actividades no necesita trabajar de abajo hacia arriba, como un algoritmo de programación dinámica basado en tablas. En su lugar, puede funcionar de arriba hacia abajo, eligiendo una actividad para colocarla en la solución óptima y luego resolviendo el subproblema de elegir actividades entre aquellas que son compatibles con las ya elegidas. Los algoritmos codiciosos suelen tener este diseño de arriba hacia abajo: hacer una elección y luego resolver un subproblema, en lugar de la técnica de abajo hacia arriba de resolver subproblemas antes de tomar una decisión.

**Un algoritmo codicioso recursivo**

Ahora que hemos visto cómo eludir el enfoque de programación dinámica y, en su lugar, usar un algoritmo codicioso de arriba hacia abajo, podemos escribir un procedimiento recursivo directo para resolver el problema de selección de actividad. El procedimiento ACTIVITY-SELECTOR RECURSIVO toma los tiempos de inicio y fin de las actividades, representados como arrays  $s$  y  $f$ , el índice  $k$  que define el subproblema  $S_k$  a resolver,

---

<sup>2</sup>Debido a que el pseudocódigo toma  $s$  y  $f$  como matrices, los indexa con corchetes en lugar de subíndices.

el tamaño  $n$  del problema original. Devuelve un conjunto de tamaño máximo de actividades mutuamente compatibles en  $S_k$ . Suponemos que las  $n$  actividades de entrada ya están ordenadas por tiempo de finalización creciente monótonamente, de acuerdo con la ecuación (16.1). Si no, podemos clasificarlos en este orden en  $O(n \lg n)$  time, rompiendo empates arbitrariamente. Para comenzar, agregamos la actividad ficticia  $a_0$  con  $f_0 = 0$ , de modo que el subproblema  $S_0$  es todo el conjunto de actividades  $S$ . La llamada inicial, que resuelve todo el problema, es **RECURSIVE-ACTIVITY-SELECTOR.s; F; 0; norte/**.

```

RECURSIVO-ACTIVIDAD-SELECTOR.s; F; k; n/ 1 m D
k C 1 2 while mn
and sŒEm < f Œk 3 m D m C 1 4 if mn 5           // encuentra la primera actividad en Sk para terminar
return famg [ RECURSIVE-
ACTIVITY-
SELECTOR.s; F; metro; n/ 6 más volver ;

```

La figura 16.1 muestra el funcionamiento del algoritmo. En una llamada recursiva dada **RECURSIVE-ACTIVITY-SELECTOR.s; F; k; n/**, el ciclo while de las líneas 2–3 busca la primera actividad en  $S_k$  para finalizar. El ciclo examina  $a_k C_1; a_k C_2; \dots; a_n$ , hasta encontrar la primera actividad  $a_m$  que sea compatible con  $a_k$ ; tal actividad tiene  $s_m \leq f_k$ . Si el bucle termina porque encuentra tal actividad, la línea 5 devuelve la unión de  $famg$  y el subconjunto de tamaño máximo de  $S_m$  devuelto por la llamada recursiva **RECURSIVE-ACTIVITY-SELECTOR.s; F; metro; norte/**. Alternativamente, el ciclo puede terminar porque  $m > n$ , en cuyo caso hemos examinado todas las actividades en  $S_k$  sin encontrar ninguna que sea compatible con  $a_k$ . En este caso,  $S_k = \emptyset$ , por lo que el procedimiento devuelve  $\emptyset$ ; en la línea 6.

Suponiendo que las actividades ya han sido ordenadas por tiempos de finalización, el tiempo de ejecución de la llamada **RECURSIVE-ACTIVITY-SELECTOR.s; F; 0; n/** es  $O(n^2)$ , que podemos ver de la siguiente manera. En todas las llamadas recursivas, cada actividad se examina exactamente una vez en la prueba de bucle while de la línea 2. En particular, la actividad  $a_i$  se examina en la última llamada realizada en la que  $k < i$ .

#### Un algoritmo codicioso iterativo

Fácilmente podemos convertir nuestro procedimiento recursivo a uno iterativo. El procedimiento **RECURSIVE-ACTIVITY-SELECTOR** es casi "recursivo de cola" (vea el problema 7-4): termina con una llamada recursiva a sí mismo seguida de una operación de unión. Por lo general, es una tarea sencilla transformar un procedimiento recursivo de cola en una forma iterativa; de hecho, algunos compiladores para ciertos lenguajes de programación realizan esta tarea automáticamente. Tal como está escrito, **RECURSIVE-ACTIVITY-SELECTOR** funciona para subproblemas  $S_k$ , es decir, subproblemas que consisten en las últimas actividades en terminar.

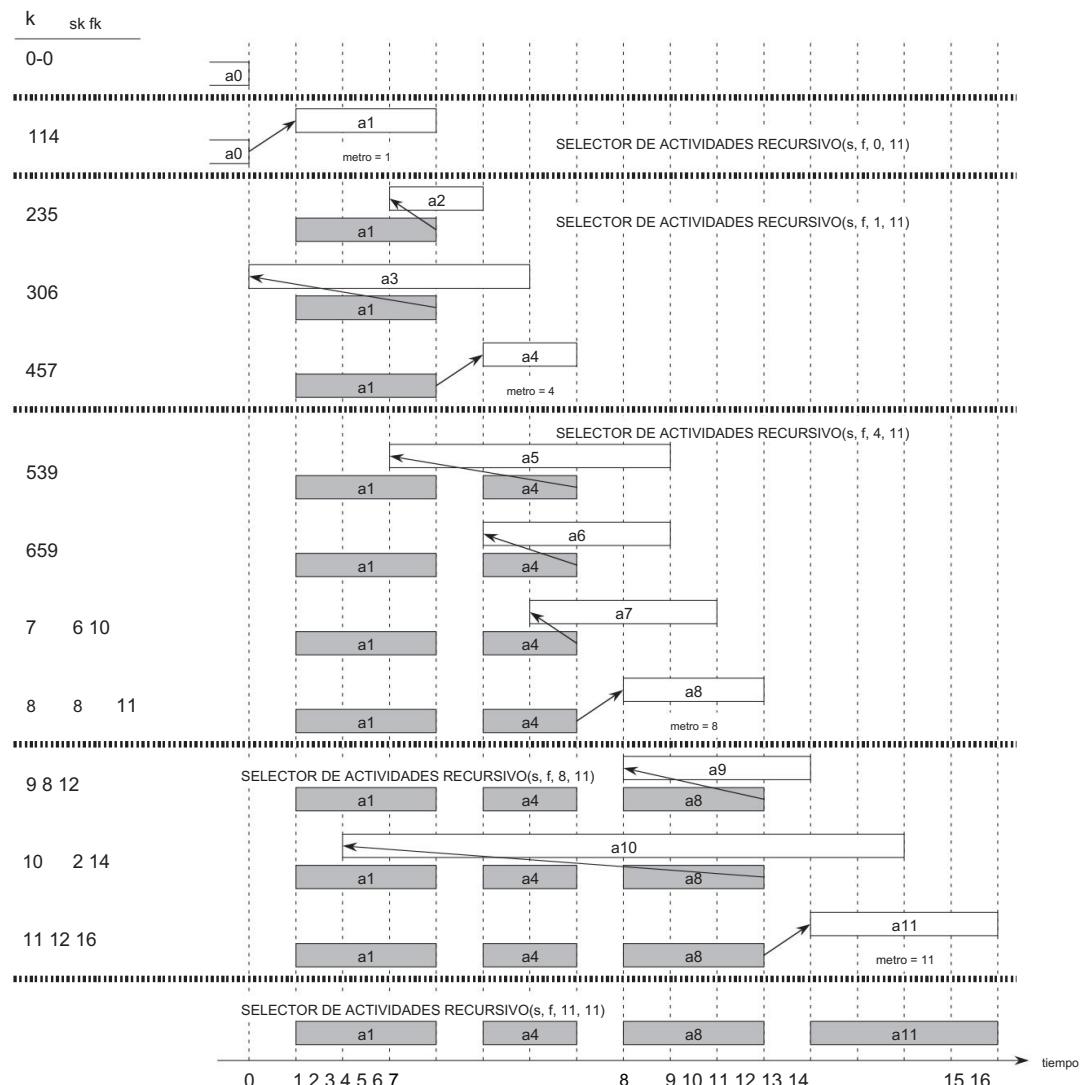


Figura 16.1 La operación de RECURSIVE-ACTIVITY-SELECTOR en las 11 actividades dadas anteriormente. Las actividades consideradas en cada llamada recursiva aparecen entre líneas horizontales. La actividad ficticia a0 finaliza en el tiempo 0, y la llamada inicial RECURSIVE-ACTIVITY-SELECTOR.s; F; 0; 11/, selecciona la actividad a1. En cada llamada recursiva, se sombrean las actividades que ya han sido seleccionadas, y se está considerando la actividad que se muestra en blanco. Si la hora de inicio de una actividad es anterior a la hora de finalización de la última actividad añadida (la flecha entre ambas apunta hacia la izquierda), se rechaza. De lo contrario (la flecha apunta directamente hacia arriba o hacia la derecha), se selecciona. La última llamada recursiva, RECURSIVE-ACTIVITY-SELECTOR.s; F; 11; 11/, vuelve ;. El conjunto resultante de actividades seleccionadas es fa1; a4; a8; a11g.

El procedimiento GREEDY-ACTIVITY-SELECTOR es una versión iterativa del procedimiento RECURSIVE-ACTIVITY-SELECTOR. También supone que las actividades de entrada están ordenadas por un tiempo de finalización que aumenta monótonamente. Reúne las actividades seleccionadas en un conjunto A y devuelve este conjunto cuando termina.

CODICIOSO-ACTIVIDAD-SELECTOR.s; f / 1 n

```
D s:longitud 2 AD
fa1g 3 k D 1 4
for m D 2 to
n 5 if sŒEm f Œk ADA
[ famg k D m
6
7
8 vuelta A
```

El procedimiento funciona de la siguiente manera. La variable k indexa la adición más reciente a A, correspondiente a la actividad ak en la versión recursiva. Dado que consideramos las actividades en orden de tiempo de finalización monótonamente creciente, fk es siempre el tiempo máximo de finalización de cualquier actividad en A. Es decir,

fk D max ffi W ai 2 Ag : (16.3)

Las líneas 2 y 3 seleccionan la actividad a1, inicializan A para que contenga solo esta actividad e inicializan k para indexar esta actividad. El ciclo for de las líneas 4 a 7 encuentra la actividad más temprana en Sk para terminar. El ciclo considera cada actividad am a su vez y agrega am a A si es compatible con todas las actividades previamente seleccionadas; tal actividad es la primera en terminar en Sk . Para ver si la actividad am es compatible con todas las actividades actualmente en A, basta con comprobar (en la línea 5) mediante la ecuación (16.3) que su hora de inicio sm no sea anterior a la hora de finalización fk de la actividad añadida más recientemente a A. Si la actividad am es compatible, entonces las líneas 6 y 7 suman la actividad am a A y establecen k en m. El conjunto A devuelto por la llamada GREEDY-ACTIVITY-SELECTOR.s; f/ es precisamente el conjunto devuelto por la llamada RECURSIVE-ACTIVITY-SELECTOR.s; F; 0; norte/.

Al igual que la versión recursiva, GREEDY-ACTIVITY-SELECTOR programa un conjunto de n actividades en „n/ tiempo, asumiendo que las actividades ya fueron ordenadas inicialmente por sus tiempos de finalización.

## Ejercicios

### 16.1-1

Proporcione un algoritmo de programación dinámica para el problema de selección de actividades, basado en la recurrencia (16.2). Haga que su algoritmo calcule los tamaños cŒi; j como se definió anteriormente y también producen el subconjunto de tamaño máximo de actividades mutuamente compatibles.

Suponga que las entradas se han ordenado como en la ecuación (16.1). Compare el tiempo de ejecución de su solución con el tiempo de ejecución de GREEDY-ACTIVITY-SELECTOR.

#### 16.1-2

Supongamos que en lugar de seleccionar siempre la primera actividad para terminar, seleccionamos la última actividad para comenzar que sea compatible con todas las actividades seleccionadas anteriormente. Describa cómo este enfoque es un algoritmo codicioso y demuestre que produce una solución óptima.

#### 16.1-3

No cualquier enfoque codicioso del problema de selección de actividades produce un conjunto de tamaño máximo de actividades mutuamente compatibles. Dé un ejemplo para mostrar que el enfoque de seleccionar la actividad de menor duración entre aquellas que son compatibles con las actividades previamente seleccionadas no funciona. Haga lo mismo con los enfoques de seleccionar siempre la actividad compatible que se superpone con la menor cantidad de otras actividades restantes y seleccionar siempre la actividad restante compatible con la hora de inicio más temprana.

#### 16.1-4

Supongamos que tenemos un conjunto de actividades para programar entre un gran número de aulas, donde cualquier actividad puede tener lugar en cualquier aula. Deseamos programar todas las actividades utilizando la menor cantidad de salas de conferencias posible. Proporcione un algoritmo codicioso eficiente para determinar qué actividad debe usar qué sala de conferencias.

(Este problema también se conoce como el problema de coloreado del gráfico de intervalo. Podemos crear un gráfico de intervalo cuyos vértices sean las actividades dadas y cuyos bordes conecten actividades incompatibles. El menor número de colores requerido para colorear cada vértice de modo que no haya dos vértices adyacentes el mismo color corresponde a encontrar la menor cantidad de salas de conferencias necesarias para programar todas las actividades dadas).

#### 16.1-5

Considere una modificación al problema de selección de actividades en el que cada actividad  $a_i$  tiene, además de un tiempo de inicio y finalización, un valor. El objetivo ya no es maximizar el número de actividades programadas, sino maximizar el valor total de las actividades programadas. Es decir, deseamos elegir un conjunto  $A$  de actividades compatibles tal que  $P$  se maximice. Proporcione un algoritmo de tiempo polinomial para  $\text{ak2A}$  k este problema.

---

## 16.2 Elementos de la estrategia codiciosa

Un algoritmo codicioso obtiene una solución óptima a un problema haciendo una secuencia de elecciones. En cada punto de decisión, el algoritmo elige lo que parece mejor en ese momento. Esta estrategia heurística no siempre produce una solución óptima, pero como vimos en el problema de selección de actividades, a veces lo hace. En esta sección se analizan algunas de las propiedades generales de los métodos codiciosos.

El proceso que seguimos en la Sección 16.1 para desarrollar un algoritmo codicioso fue un poco más involucrado de lo que es típico. Pasamos por los siguientes pasos:

1. Determinar la subestructura óptima del problema.
2. Desarrollar una solución recursiva. (Para el problema de selección de actividad, formulamos la recurrencia (16.2), pero pasamos por alto el desarrollo de un algoritmo recursivo basado en esta recurrencia).
3. Muestre que si tomamos la decisión codiciosa, entonces solo queda un subproblema.
4. Demostrar que siempre es seguro tomar la decisión codiciosa. (Los pasos 3 y 4 pueden ocurrir en cualquier orden).
5. Desarrolle un algoritmo recursivo que implemente la estrategia codiciosa.
6. Convierta el algoritmo recursivo en un algoritmo iterativo.

Al seguir estos pasos, vimos con gran detalle la programación dinámica que subyace a un algoritmo codicioso. Por ejemplo, en el problema de selección de actividades, primero definimos los subproblemas  $S_{ij}$ , donde tanto  $i$  como  $j$  variaban. Entonces descubrimos que si siempre hacíamos la elección codiciosa, podíamos restringir los subproblemas para que fueran de la forma  $S_k$ .

Alternativamente, podríamos haber diseñado nuestra subestructura óptima con una elección codiciosa en mente, de modo que la elección dejase solo un subproblema por resolver. En el problema de selección de actividades, podríamos haber comenzado eliminando el segundo subíndice y definiendo subproblemas de la forma  $S_k$ . Entonces, podríamos haber probado que una elección codiciosa (la primera actividad  $a_m$  para terminar en  $S_k$ ), combinada con una solución óptima para el conjunto restante  $S_m$  de actividades compatibles, produce una solución óptima para  $S_k$ . De manera más general, diseñamos algoritmos voraces de acuerdo con la siguiente secuencia de pasos:

1. Plantee el problema de optimización como uno en el que hacemos una elección y nos quedamos con un subproblema a resolver.
2. Demostrar que siempre hay una solución óptima al problema original que hace la elección codiciosa, de modo que la elección codiciosa sea siempre segura.

3. Demostrar la subestructura óptima mostrando que, habiendo hecho la elección codiciosa, lo que queda es un subproblema con la propiedad de que si combinamos una solución óptima al subproblema con la elección codiciosa que hemos hecho, llegamos a una solución óptima a la original problema.

Usaremos este proceso más directo en secciones posteriores de este capítulo. Sin embargo, debajo de cada algoritmo codicioso, casi siempre hay una solución de programación dinámica más engorrosa.

¿Cómo podemos saber si un algoritmo codicioso resolverá un problema de optimización particular? De ninguna manera funciona todo el tiempo, pero la propiedad de elección codiciosa y la subestructura óptima son los dos ingredientes clave. Si podemos demostrar que el problema tiene estas propiedades, entonces estamos bien encaminados para desarrollar un algoritmo codicioso para él.

#### Propiedad de elección codiciosa

El primer ingrediente clave es la propiedad de la elección codiciosa: podemos ensamblar una solución globalmente óptima haciendo elecciones localmente óptimas (codiciosas). En otras palabras, cuando estamos considerando qué elección hacer, hacemos la elección que se ve mejor en el problema actual, sin considerar los resultados de los subproblemas.

Aquí es donde los algoritmos codiciosos difieren de la programación dinámica. En la programación dinámica, hacemos una elección en cada paso, pero la elección generalmente depende de las soluciones a los subproblemas. En consecuencia, normalmente resolvemos problemas de programación dinámica de forma ascendente, progresando desde subproblemas más pequeños a subproblemas más grandes. (Alternativamente, podemos resolverlos de arriba hacia abajo, pero memorizando. Por supuesto, aunque el código funciona de arriba hacia abajo, aún debemos resolver los subproblemas antes de tomar una decisión). momento y luego resolver el subproblema que queda. La elección hecha por un algoritmo codicioso puede depender de las elecciones realizadas hasta el momento, pero no puede depender de ninguna elección futura o de las soluciones a los subproblemas. Así, a diferencia de la programación dinámica, que resuelve los subproblemas antes de hacer la primera elección, un algoritmo codicioso hace su primera elección antes de resolver cualquier subproblema. Un algoritmo de programación dinámica procede de abajo hacia arriba, mientras que una estrategia codiciosa generalmente progresiva de arriba hacia abajo, haciendo una elección codiciosa tras otra, reduciendo cada instancia dada del problema a una más pequeña.

Por supuesto, debemos demostrar que una elección codiciosa en cada paso produce una solución globalmente óptima. Típicamente, como en el caso del Teorema 16.1, la prueba examina una solución óptima global para algún subproblema. Luego muestra cómo modificar la solución para sustituir la opción codiciosa por alguna otra opción, lo que da como resultado un subproblema similar, pero más pequeño.

Por lo general, podemos hacer la elección codiciosa de manera más eficiente que cuando tenemos que considerar un conjunto más amplio de opciones. Por ejemplo, en el problema de selección de actividades, as-

suponiendo que ya habíamos clasificado las actividades en un orden monótonamente creciente de tiempos de finalización, necesitábamos examinar cada actividad solo una vez. Al preprocessar la entrada o al usar una estructura de datos adecuada (a menudo una cola de prioridad), a menudo podemos tomar decisiones codiciosas rápidamente, lo que genera un algoritmo eficiente.

### Subestructura óptima

Un problema exhibe una subestructura óptima si una solución óptima al problema contiene soluciones óptimas a los subproblemas. Esta propiedad es un ingrediente clave para evaluar la aplicabilidad de la programación dinámica, así como los algoritmos codiciosos. Como ejemplo de subestructura óptima, recuerde cómo demostramos en la Sección 16.1 que si una solución óptima al subproblema  $S_{ij}$  incluye una actividad  $a_k$ , entonces también debe contener soluciones óptimas a los subproblemas  $S_{ik}$  y  $S_{kj}$ . Dada esta subestructura óptima, argumentamos que si supiéramos qué actividad usar como  $a_k$ , podríamos construir una solución óptima para  $S_{ij}$  seleccionando  $a_k$  junto con todas las actividades en soluciones óptimas a los subproblemas  $S_{ik}$  y  $S_{kj}$ . Con base en esta observación de la subestructura óptima, pudimos idear la recursión (16.2) que describía el valor de una solución óptima.

Usualmente usamos un enfoque más directo con respecto a la subestructura óptima cuando lo aplicamos a algoritmos codiciosos. Como se mencionó anteriormente, podemos darnos el lujo de suponer que llegamos a un subproblema al haber tomado la decisión codiciosa en el problema original. Todo lo que realmente necesitamos hacer es argumentar que una solución óptima al subproblema, combinada con la elección codiciosa ya hecha, produce una solución óptima al problema original. Este esquema usa implícitamente la inducción en los subproblemas para demostrar que hacer la elección codiciosa en cada paso produce una solución óptima.

### Programación codiciosa versus dinámica

Debido a que tanto la estrategia codiciosa como la de programación dinámica explotan una subestructura óptima, puede verse tentado a generar una solución de programación dinámica a un problema cuando una solución codiciosa es suficiente o, por el contrario, puede pensar erróneamente que una solución codiciosa funciona cuando en realidad se requiere una solución de programación dinámica. Para ilustrar las sutilidades entre las dos técnicas, investiguemos dos variantes de un problema de optimización clásico.

El problema de la mochila 0-1 es el siguiente. Un ladrón que asalta una tienda encuentra  $n$  artículos. El  $i$ -ésimo artículo vale dólares y pesa  $w_i$  libras, donde  $y$   $w_i$  son números enteros. El ladrón quiere llevarse una carga lo más valiosa posible, pero puede llevar como mucho  $W$  libras en su mochila, por algún número entero  $W$ . ¿Qué elementos debe tomar?

(A esto lo llamamos el problema de la mochila 0-1 porque para cada artículo, el ladrón debe

tómalo o déjalo atrás; no puede tomar una cantidad fraccionaria de un artículo o tomar un artículo más de una vez).

En el problema de la mochila fraccionada, la configuración es la misma, pero el ladrón puede tomar fracciones de artículos, en lugar de tener que hacer una elección binaria (0-1) para cada artículo. Puedes pensar en un artículo en el problema de la mochila 0-1 como si fuera un lingote de oro y un artículo en el problema de la mochila fraccional como más como polvo de oro.

Ambos problemas de mochila exhiben la propiedad de subestructura óptima. Para el problema 0-1, considere la carga más valiosa que pesa como máximo  $W$  libras. Si quitamos el artículo  $j$  de esta carga, la carga restante debe ser la carga más valiosa que pesa como máximo  $W - w_j$  que el ladrón puede tomar de los  $n - 1$  artículos originales excluyendo  $j$ . Para el problema fraccionario comparable, considere que si quitamos un peso  $w$  de un artículo  $j$  de la carga óptima, la carga restante debe ser la carga más valiosa que pesa como máximo  $W - w$  que el ladrón puede tomar de los  $n - 1$  artículos originales más  $w_j$  libras del artículo  $j$ .

Aunque los problemas son similares, podemos resolver el problema de la mochila fraccional con una estrategia codiciosa, pero no podemos resolver el problema 0-1 con esa estrategia. Para resolver el problema fraccionario, primero calculamos el valor por libra  $i = w_i / v_i$  para cada artículo. Obedeciendo a una estrategia codiciosa, el ladrón comienza por tomar la mayor cantidad posible del artículo con el mayor valor por libra. Si el suministro de ese artículo se agota y aún puede llevar más, toma la mayor cantidad posible del artículo con el siguiente mayor valor por libra, y así sucesivamente, hasta que alcanza su límite de peso  $W$ . Por lo tanto, al clasificar los artículos por valor por libra, el algoritmo codicioso se ejecuta en  $O(n \lg n)$  time. La prueba de que el problema fraccionario de la mochila tiene la propiedad de elección codiciosa la dejamos como Ejercicio 16.2-1.

Para ver que esta estrategia codiciosa no funciona para el problema de la mochila 0-1, considere la instancia del problema que se ilustra en la figura 16.2(a). Este ejemplo tiene 3 artículos y una mochila que puede contener 50 libras. El artículo 1 pesa 10 libras y vale 60 dólares. El artículo 2 pesa 20 libras y vale 100 dólares. El artículo 3 pesa 30 libras y vale 120 dólares. Por lo tanto, el valor por libra del artículo 1 es de 6 dólares por libra, que es mayor que el valor por libra del artículo 2 (5 dólares por libra) o del artículo 3 (4 dólares por libra). La estrategia codiciosa, por lo tanto, tomaría el elemento 1 primero. Sin embargo, como puede ver en el análisis de caso de la figura 16.2(b), la solución óptima toma los elementos 2 y 3, dejando atrás el elemento 1. Las dos soluciones posibles que toman el ítem 1 son ambas subóptimas.

Sin embargo, para el problema fraccionario comparable, la estrategia codiciosa, que toma el elemento 1 primero, produce una solución óptima, como se muestra en la figura 16.2(c). Tomar el artículo 1 no funciona en el problema 0-1 porque el ladrón no puede llenar su mochila a su máxima capacidad y el espacio vacío reduce el valor efectivo por libra de su carga. En el problema 0-1, cuando consideramos si incluir un artículo en la mochila, debemos comparar la solución del subproblema que incluye el artículo con la solución del subproblema que excluye el artículo antes de que podamos hacer la

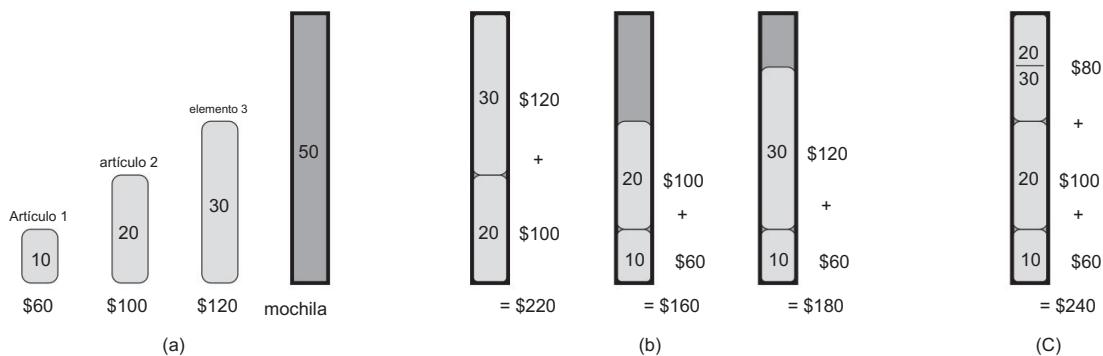


Figura 16.2 Un ejemplo que muestra que la estrategia codicosa no funciona para el problema de la mochila 0-1. (a) El ladrón debe seleccionar un subconjunto de los tres artículos que se muestran cuyo peso no debe exceder las 50 libras. (b) El subconjunto óptimo incluye los artículos 2 y 3. Cualquier solución con el artículo 1 es subóptima, aunque el artículo 1 tenga el mayor valor por libra. (c) Para el problema fraccionario de la mochila, tomar los artículos en orden de mayor valor por libra produce una solución óptima.

elección. El problema formulado de esta manera da lugar a muchos subproblemas superpuestos, un sello distintivo de la programación dinámica y, de hecho, como se le pide que muestre en el ejercicio 16.2-2, podemos usar la programación dinámica para resolver el problema 0-1.

### Ejercicios

#### 16.2-1

Demuestre que el problema fraccionario de la mochila tiene la propiedad de elección codicosa.

#### 16.2-2

Proporcione una solución de programación dinámica al problema de la mochila 0-1 que se ejecuta en  $O(nW)$ , donde  $n$  es el número de artículos y  $W$  es el peso máximo de artículos que el ladrón puede poner en su mochila.

#### 16.2-3

Suponga que en un problema de mochila 0-1, el orden de los artículos cuando se ordenan por peso creciente es el mismo que cuando se ordenan por valor decreciente. Proporcione un algoritmo eficiente para encontrar una solución óptima a esta variante del problema de la mochila y argumente que su algoritmo es correcto.

#### 16.2-4

El profesor Gekko siempre ha soñado con patinar en línea en Dakota del Norte. Planea cruzar el estado por la autopista US 2, que va desde Grand Forks, en la frontera este con Minnesota, hasta Williston, cerca de la frontera oeste con Montana.

El profesor puede llevar dos litros de agua y puede patinar  $m$  millas antes de quedarse sin agua. (Debido a que Dakota del Norte es relativamente plana, el profesor no tiene que preocuparse por beber agua a un ritmo mayor en las secciones cuesta arriba que en las secciones planas o cuesta abajo). El profesor comenzará en Grand Forks con dos litros completos de agua. Su mapa oficial del estado de Dakota del Norte muestra todos los lugares a lo largo de la US 2 en los que puede recargar su agua y las distancias entre estos lugares.

El objetivo del profesor es minimizar el número de paradas de agua a lo largo de su ruta por el estado. Proporcione un método eficiente mediante el cual pueda determinar qué paradas de agua debe hacer. Demuestre que su estrategia produce una solución óptima e indique su tiempo de ejecución.

#### 16.2-5

Describa un algoritmo eficiente que, dado un conjunto  $x_1; x_2; \dots; x_n$  de puntos en la línea real, determina el conjunto más pequeño de intervalos cerrados de longitud unitaria que contiene todos los puntos dados. Argumenta que tu algoritmo es correcto.

#### 16.2-6 ?

Muestre cómo resolver el problema fraccionario de la mochila en  $O(n/t)$ .

#### 16.2-7

Suponga que le dan dos conjuntos A y B, cada uno de los cuales contiene  $n$  enteros positivos. Puede elegir reordenar cada conjunto como deseé. Después de reordenar, sea  $a_i$  el  $i$ -ésimo elemento del conjunto A y sea  $b_i$  el  $i$ -ésimo elemento del conjunto B. Luego recibe un pago de  $Q_n b_i$ . Proporcione un algoritmo que maximizará su pago. Demuestre que su algoritmo ID1 maximiza el pago y establezca su tiempo de ejecución.

### 16.3 Códigos de Huffman

Los códigos de Huffman comprimen datos de manera muy efectiva: son típicos ahorros del 20% al 90%, dependiendo de las características de los datos que se comprimen. Consideramos que los datos son una secuencia de caracteres. El algoritmo codicioso de Huffman utiliza una tabla que indica la frecuencia con la que aparece cada carácter (es decir, su frecuencia) para construir una forma óptima de representar cada carácter como una cadena binaria.

Supongamos que tenemos un archivo de datos de 100.000 caracteres que deseamos almacenar de forma compacta. Observamos que los caracteres en el archivo ocurren con las frecuencias dadas por la figura 16.3. Es decir, solo aparecen 6 caracteres diferentes y el carácter a aparece 45.000 veces.

Tenemos muchas opciones sobre cómo representar dicho archivo de información. Aquí, consideraremos el problema de diseñar un código de caracteres binarios (o código para abbreviar)

	a	B	C	D	e	F
Frecuencia (en miles)	45	13	9	000	001	2
Palabra clave de longitud fija	101	101	100	111	1101	1100
Palabra de código de longitud variable 0						5

Figura 16.3 Problema de codificación de caracteres. Un archivo de datos de 100 000 caracteres contiene solo los caracteres a–f, con las frecuencias indicadas. Si asignamos a cada carácter una palabra clave de 3 bits, podemos codificar el archivo en 300.000 bits. Usando el código de longitud variable que se muestra, podemos codificar el archivo en solo 224,000 bits.

en el que cada carácter está representado por una cadena binaria única, a la que llamamos palabra clave. Si usamos un código de longitud fija, necesitamos 3 bits para representar 6 caracteres: f = 101. Este archivo. ¿Podemos hacerlo      método requiere 300.000 bits para codificar a = 000, b = 001, ..., todo el mejor?

Un código de longitud variable puede funcionar considerablemente mejor que un código de longitud fija, dando palabras de código cortas a los caracteres frecuentes y palabras de código largas a los caracteres poco frecuentes. La figura 16.3 muestra un código de este tipo; aquí la cadena de 1 bit 0 representa a, y la cadena de 4 bits 1100 representa f. Este código requiere

.45 1 C 13 3 C 12 3 C 16 3 C 9 4 C 5 4/ 1,000 D 224,000 bits

para representar el expediente, un ahorro aproximado del 25%. De hecho, este es un código de caracteres óptimo para este archivo, como veremos.

#### Códigos de prefijo

Consideramos aquí solo códigos en los que ninguna palabra clave es también un prefijo de alguna otra palabra clave. Estos códigos se denominan códigos de prefijo<sup>3</sup>Aunque no lo demostraremos aquí, un código de prefijo siempre puede lograr la compresión de datos óptima entre cualquier código de carácter, por lo que no sufrimos ninguna pérdida de generalidad al restringir nuestra atención a los códigos de prefijo.

La codificación siempre es simple para cualquier código de carácter binario; simplemente concatenamos las palabras clave que representan cada carácter del archivo. Por ejemplo, con el código de prefijo de longitud variable de la figura 16.3, codificamos el archivo de 3 caracteres abc como 0101100 D 0101100, donde "" denota concatenación.

Los códigos de prefijo son deseables porque simplifican la decodificación. Dado que ninguna palabra clave es un prefijo de otra, la palabra clave que comienza un archivo codificado no es ambigua. Simplemente podemos identificar la palabra clave inicial, traducirla de nuevo al carácter original

---

<sup>3</sup>Quizás “códigos sin prefijo” sería un mejor nombre, pero el término “códigos de prefijo” es estándar en el literatura.

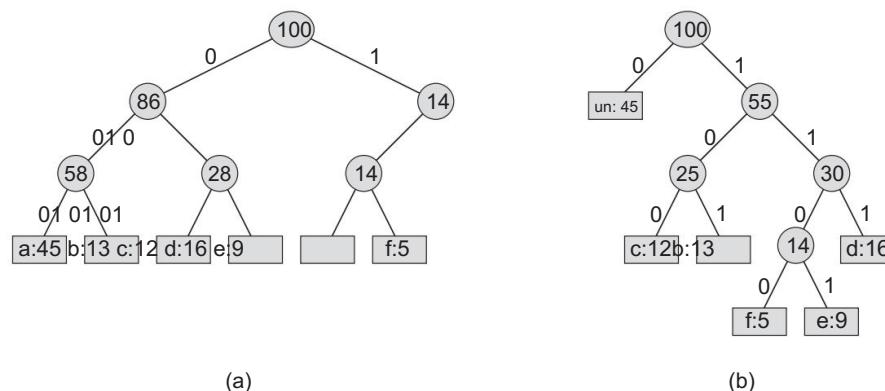


Figura 16.4 Árboles correspondientes a los esquemas de codificación de la Figura 16.3. Cada hoja está etiquetada con un carácter y su frecuencia de aparición. Cada nodo interno está etiquetado con la suma de las frecuencias de las hojas en su subárbol. (a) El árbol correspondiente al código de longitud fija  $a = 000, \dots, f = 101$ . (b) El árbol correspondiente al código de prefijo óptimo  $a = 0, b = 101, \dots, f = 1100$ .

carácter, y repita el proceso de decodificación en el resto del archivo codificado. En nuestro ejemplo, la cadena 001011101 se analiza únicamente como 0 0 101 1101, que se decodifica en aabe.

El proceso de decodificación necesita una representación conveniente para el código de prefijo para que podamos seleccionar fácilmente la palabra clave inicial. Un árbol binario cuyas hojas son los caracteres dados proporciona una representación de este tipo. Interpretamos la palabra clave binaria para un carácter como la ruta simple desde la raíz hasta ese carácter, donde 0 significa "ir al elemento secundario de la izquierda" y 1 significa "ir al elemento secundario de la derecha". La Figura 16.4 muestra los árboles para los dos códigos de nuestro ejemplo. Tenga en cuenta que estos no son árboles de búsqueda binarios, ya que las hojas no necesitan aparecer ordenadas y los nodos internos no contienen claves de caracteres.

Un código óptimo para un archivo siempre se representa mediante un árbol binario completo , en el que cada nodo no hoja tiene dos hijos (vea el ejercicio 16.3-2). El código de longitud fija de nuestro ejemplo no es óptimo ya que su árbol, que se muestra en la figura 16.4(a), no es un árbol binario completo: contiene palabras de código que comienzan con 10. . . , pero ninguno que comience 11. . . . Como ahora podemos restringir nuestra atención a árboles binarios completos, podemos decir que si  $C$  es el alfabeto del que se extraen los caracteres y todas las frecuencias de caracteres son positivas, entonces el árbol para un código de prefijo óptimo tiene exactamente  $|C|$  hojas, una para cada letra del alfabeto, y exactamente  $|C| - 1$  nodos internos (vea el Ejercicio B.5-3).

Dado un árbol T correspondiente a un código de prefijo, podemos calcular fácilmente el número de bits necesarios para codificar un archivo. Para cada carácter c en el alfabeto C, deje que el atributo  $c:\text{freq}$  denote la frecuencia de c en el archivo y deje que  $d_T(c)$  denote la profundidad

de la hoja de c en el árbol. Tenga en cuenta que  $d_T \cdot c$  también es la longitud de la palabra clave para el carácter c. Por tanto, el número de bits necesarios para codificar un archivo es

BT / DX c:frecuencia dT .c/ ;  
c2c

que definimos como el costo del árbol T .

### Construyendo un código de Huffman

Huffman inventó un algoritmo codicioso que construye un código de prefijo óptimo llamado código Huffman. De acuerdo con nuestras observaciones en la Sección 16.2, su prueba de corrección se basa en la propiedad de elección codiciosa y la subestructura óptima. En lugar de demostrar que estas propiedades se mantienen y luego desarrollar el pseudocódigo, presentamos primero el pseudocódigo. Si lo hace, ayudará a aclarar cómo el algoritmo toma decisiones codiciosas.

En el pseudocódigo que sigue, asumimos que C es un conjunto de n caracteres y que cada carácter c  $\in C$  es un objeto con un atributo c.freq que da su frecuencia.

El algoritmo construye el árbol T correspondiente al código óptimo de forma ascendente. Comienza con un conjunto de hojas  $jC_j$  y realiza una secuencia de operaciones de " fusión"  $jC_j$  1 para crear el árbol final. El algoritmo utiliza una cola Q de prioridad mínima, codificada en el atributo freq , para identificar los dos objetos menos frecuentes para fusionarlos. Cuando fusionamos dos objetos, el resultado es un nuevo objeto cuya frecuencia es la suma de las frecuencias de los dos objetos que se fusionaron.

### HUFFMAN.C / 1

```

n D jCj 2 QDC
3 para i D 1 a
n 1 4
      asignar un nuevo nodo '
5      ':izquierda D x D EXTRACT-MIN.Q/
6      ':derecha D y D EXTRACT-MIN.Q/ ':freq D
7      x.freq C y.freq INSERT.Q; '/ 9
8      return EXTRACT-
MIN.Q/ // devuelve la raíz del árbol

```

Para nuestro ejemplo, el algoritmo de Huffman procede como se muestra en la figura 16.5. Dado que el alfabeto contiene 6 letras, el tamaño inicial de la cola es n D 6 y 5 pasos de combinación construyen el árbol. El árbol final representa el código de prefijo óptimo. La palabra clave para una letra es la secuencia de etiquetas de borde en el camino simple desde la raíz hasta la letra.

La línea 2 inicializa la cola de prioridad mínima Q con los caracteres en C. El bucle for en las líneas 3 a 8 extrae repetidamente los dos nodos x e y de menor frecuencia

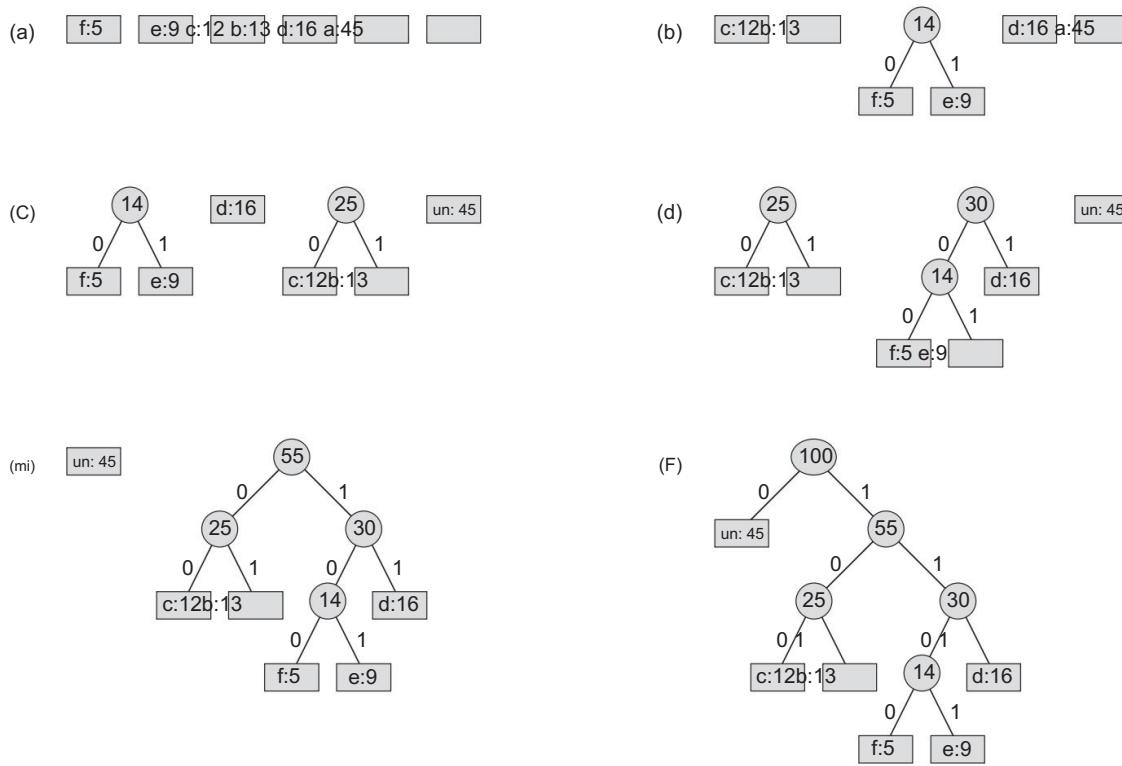


Figura 16.5 Los pasos del algoritmo de Huffman para las frecuencias dadas en la Figura 16.3. Cada parte muestra el contenido de la cola ordenado en orden creciente por frecuencia. En cada paso, los dos árboles con las frecuencias más bajas se fusionan. Las hojas se muestran como rectángulos que contienen un carácter y su frecuencia. Los nodos internos se muestran como círculos que contienen la suma de las frecuencias de sus hijos. Una arista que conecta un nodo interno con sus hijos se etiqueta con 0 si es una arista con un hijo izquierdo y con 1 si es una arista con un hijo derecho. La palabra clave de una letra es la secuencia de etiquetas en los bordes que conectan la raíz con la hoja de esa letra. (a) El conjunto inicial de 6 nodos, uno para cada letra. (b)–(e) Etapas intermedias. (f) El último árbol.

de la cola, reemplazándolos en la cola con un nuevo nodo que representa su fusión. La frecuencia de 'x' se calcula como la suma de las frecuencias de x y e en la línea 7. El nodo 'x' tiene 'y' como hijo izquierdo y 'e' como hijo derecho. (Este orden es arbitrario; cambiar el hijo izquierdo y derecho de cualquier nodo produce un código diferente del mismo costo). Después de n-1 fusiones, la línea 9 devuelve el único nodo que queda en la cola, que es la raíz del árbol de código.

Aunque el algoritmo produciría el mismo resultado si elimináramos las variables x e y, asignando directamente a 'x:left' y 'x:right' en las líneas 5 y 6, y cambiando la línea 7 a 'x:freq D x:left:freq C x:right:freq'—usaremos el nodo

nombres x e y en la prueba de corrección. Por ello, nos parece conveniente dejarlos dentro.

Para analizar el tiempo de ejecución del algoritmo de Huffman, asumimos que Q se implementa como un montón mínimo binario (consulte el Capítulo 6). Para un conjunto C de n caracteres, podemos inicializar Q en la línea 2 en  $O(n \lg n)$  usando el procedimiento BUILD-MIN-HEAP discutido en la Sección 6.3. El ciclo for en las líneas 3 a 8 se ejecuta exactamente  $n - 1$  veces, y dado que cada operación de almacenamiento dinámico requiere un tiempo  $O(\lg n)$ , el ciclo contribuye con  $O(n \lg n)$  al tiempo de ejecución. Por tanto, el tiempo total de ejecución de HUFFMAN en un conjunto de n caracteres es  $O(n \lg n)$ . Podemos reducir el tiempo de ejecución a  $O(n \lg \lg n)$  reemplazando el min-heap binario con un árbol de van Emde Boas (vea el Capítulo 20).

#### Corrección del algoritmo de Huffman

Para probar que el algoritmo voraz de HUFFMAN es correcto, mostramos que el problema de determinar un código de prefijo óptimo exhibe las propiedades de elección voraz y de subestructura óptima. El siguiente lema muestra que se cumple la propiedad de elección codiciosa.

#### Lema 16.2 Sea

C un alfabeto en el que cada carácter c  $\in C$  tiene una frecuencia  $c:\text{freq}$ . Sean x e y dos caracteres en C que tienen las frecuencias más bajas. Entonces existe un código de prefijo óptimo para C en el que las palabras de código para x e y tienen la misma longitud y difieren solo en el último bit.

Prueba La idea de la prueba es tomar el árbol T que representa un código de prefijo óptimo arbitrario y modificarlo para hacer un árbol que represente otro código de prefijo óptimo de modo que los caracteres x e y aparezcan como hojas hermanas de máxima profundidad en el nuevo árbol. Si podemos construir un árbol de este tipo, entonces las palabras de código para x e y tendrán la misma longitud y solo se diferenciarán en el último bit.

Sean ayb dos caracteres que son hojas hermanas de máxima profundidad en T Sin pérdida de generalidad, asumimos que  $a:\text{freq} < b:\text{freq}$  y  $x:\text{freq} < y:\text{freq}$ .

Como  $x:\text{freq}$  e  $y:\text{freq}$  son las dos frecuencias hoja más bajas, en orden, y  $a:\text{freq} < b:\text{freq}$  son dos frecuencias arbitrarias, en orden, tenemos  $x:\text{freq} < a:\text{freq} < y:\text{freq} < b:\text{freq}$ .

En el resto de la demostración, es posible que tengamos  $x:\text{freq} < D$  o  $y:\text{freq} < D$ . Sin embargo, si tuviéramos  $x:\text{freq} < D < b:\text{freq}$ , entonces también tendríamos  $a:\text{freq} < D < b:\text{freq}$  o  $x:\text{freq} < D < y:\text{freq}$  (vea el ejercicio 16.3-1), y el lema sería trivialmente verdadero. Así, supondremos que  $x:\text{freq} \leq D < b:\text{freq}$ , lo que significa que  $x \leq D < b$ .

Como muestra la figura 16.6, intercambiamos las posiciones en T de a y x para producir un árbol  $T'$ , y luego intercambiamos las posiciones en  $T'$  de b e y para producir un árbol  $T''$ .

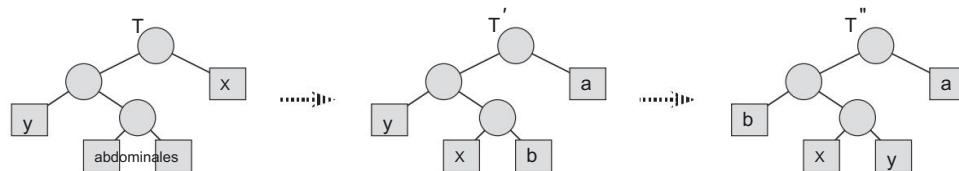


Figura 16.6 Una ilustración del paso clave en la prueba del Lema 16.2. En el árbol óptimo  $T$ , las hojas  $a$  y  $b$  son dos hermanos de máxima profundidad. Las hojas  $x$  e  $y$  son los dos caracteres con las frecuencias más bajas; aparecen en posiciones arbitrarias en  $T$ . Suponiendo que  $x \not\simeq b$ , el intercambio de hojas  $a$  y  $b$  no aumenta el costo, luego el intercambio de hojas  $b$  e  $y$  produce el árbol  $T^{00}$ . Dado que cada el costo  $x$  y  $y$  produce el árbol  $T$ , el árbol resultante  $T''$  también es un árbol óptimo.

en la que  $x$  e  $y$  son hojas hermanas de máxima profundidad. (Tenga en cuenta que si  $x \simeq b$  y  $\not\simeq a$ , entonces el árbol  $T^{00}$  pero no tiene  $x$  e  $y$  como hermanos deja la profundidad máxima. Como suponemos que  $x \not\simeq b$ , esta situación no puede ocurrir.) Por la ecuación (16.4), la diferencia de costo entre  $T$  y  $T'$  es

$$BT / BT^{00} = \frac{c_{2c}}{c_{2c}}$$

$$DX c:\text{frecuencia } d_T .c / X c:\text{frecuencia } d_T 0.c /$$

$$D x:\text{frecuencia } d_T .x / C a:\text{frecuencia } d_T .a / x:\text{frecuencia } d_T 0.x / a:\text{frecuencia } d_T$$

$$0.a / D x:\text{frecuencia } d_T .x / C a:\text{frecuencia } d_T .a / x:\text{frecuencia } d_T .a / a:\text{frecuencia } d_T .x / D .a:\text{frecuencia } x:\text{frecuencia } d_T .a / d_T .x /$$

$$0 :$$

porque tanto  $a:\text{freq } x:\text{freq}$  como  $d_T .a / d_T .x /$  son no negativos. Más específicamente,  $a:\text{freq } x:\text{freq}$  no es negativa porque  $x$  es una hoja de frecuencia mínima, y  $d_T .a / d_T .x /$  no es negativa porque  $a$  es una hoja de profundidad máxima en  $T$ . De manera similar, el intercambio de  $y$  y  $b$  no aumenta el costo  $y$ , por lo tanto,  $BT^{00} / BT$  tivo. Por tanto,  $BT^{00} / BT^{00} /$  es nonnegativo y como  $T$  es óptimo, tenemos  $BT / BT^{00}$ , lo que implica  $BT^{00} / BT /$ . Por tanto,  $T$  es un árbol óptimo en el que  $x$  e  $y$  aparecen como hojas hermanas de máxima profundidad, de las que se sigue el lema. ■

El lema 16.2 implica que el proceso de construcción de un árbol óptimo mediante fusiones puede, sin pérdida de generalidad, comenzar con la elección codiciosa de fusionar esos dos caracteres de frecuencia más baja. ¿Por qué es esta una elección codiciosa? Podemos ver el costo de una sola fusión como la suma de las frecuencias de los dos elementos que se fusionan. El ejercicio 16.3-4 muestra que el costo total del árbol construido es igual a la suma de los costos de sus fusiones. De todas las fusiones posibles en cada paso, HUFFMAN elige la que incurre en el menor costo.

El siguiente lema muestra que el problema de construir códigos de prefijos óptimos tiene la propiedad de subestructura óptima.

### Lema 16.3

Sea  $C$  un alfabeto dado con frecuencia  $c:\text{freq}$  definida para cada carácter  $c \in C$ .

Sean x e y dos caracteres en C con frecuencia mínima. Sea C0 el alfabeto C con los caracteres x e y eliminados y un nuevo carácter ' añadido, de modo que C0 DC fx;yg [ f'g. Defina f para C0 como para C, excepto que ':freq D x:freq C y:freq. Sea T cualquier árbol que represente un código de prefijo óptimo para el alfabeto C0 . Entonces, el árbol T al reemplazar el nodo hoja por ' con un nodo interno que tiene x e y como hijos, <sup>obtenido de T</sup> representa un código de prefijo óptimo para el alfabeto C.

Prueba Primero mostramos cómo expresar el costo BT / del árbol T en términos del costo BT Para cada  $\%_i$  del árbol  $T_i$ , considerando los costos de los componentes en la ecuación (16.4). carácter c  $\leq C_{fx,yg}$ , tenemos que  $dT_i \cdot c_i / D_i dT_0 \cdot c_0$ , y por lo tanto  $c_i \cdot freq(dT_i) \cdot c_i / D_i c_i \cdot freq(dT_0)$ . Como  $dT_i \cdot x_i / D_i dT_0 \cdot y_i / D_i dT_0 \leq C_1$ , tenemos  $x_i \cdot freq(dT_i) \cdot x_i / C_1 y_i \cdot freq(dT_i) \cdot y_i / D_i x_i \cdot freq(dT_i) \cdot y_i / D_i y_i \cdot freq(dT_i) \leq C_1$

de lo que concluimos que

BT / D BT o, ° / C x:frecuencia C y:frecuencia

equivalentemente,

BT ° / D BT / x:frecuencia y:frecuencia :

Probamos ahora el lema por contradicción. Supongamos que  $T$  no representa tal que envió un código de prefijo óptimo para  $C$ . Entonces existe un árbol óptimo  $T_{BT}$  de  $00^{00} < BT /$ . Sin pérdida de generalidad (por el Lema 16.2),  $T$  tiene  $x$  e  $y$  como  $coh^{00}$  el padre común hermanos. Sea  $T'$  ser el árbol  $T$  de  $x$  e  $y$  reemplazado por un  $T$  hoja  $\zeta$  con frecuencia  $freq(D) + freq(C) - freq$ . Entonces  $RT'$

000/ D BT 00/ x:freq y:freq < BT / x:freq y:freq

D\_BT produciendo una

0 / .

contradicción a la suposición de que  $T$  para  $C_0$ . Por lo tanto,  $T$  representa un código de prefijo óptimo para el alfabeto  $C$ .

Theorem 16.4

Procedimiento HUFFMAN produce un código de prefijo óptimo

Prueba Inmediata de los Lemas 16.2 y 16.3

### Ejercicios

#### 16.3-1

Explique por qué, en la demostración del Lema 16.2, si  $x:\text{freq}$  D  $b:\text{freq}$ , entonces debemos tener  $a:\text{freq}$  D  $b:\text{freq}$  D  $x:\text{freq}$  D  $y:\text{freq}$ .

#### 16.3-2

Demuestre que un árbol binario que no está lleno no puede corresponder a un código de prefijo óptimo.

#### 16.3-3

¿Cuál es un código de Huffman óptimo para el siguiente conjunto de frecuencias, basado en los primeros 8 números de Fibonacci?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

¿Puedes generalizar tu respuesta para encontrar el código óptimo cuando las frecuencias son los primeros  $n$  números de Fibonacci?

#### 16.3-4

Demuestre que también podemos expresar el costo total de un árbol para un código como la suma, sobre todos los nodos internos, de las frecuencias combinadas de los dos hijos del nodo.

#### 16.3-5

Demuestre que si ordenamos los caracteres en un alfabeto de modo que sus frecuencias disminuyan monótonamente, entonces existe un código óptimo cuyas longitudes de palabra clave aumentan monótonamente.

#### 16.3-6

Supongamos que tenemos un código de prefijo óptimo en un conjunto de CD  $f_0; 1; \dots; n$   $1g$  de caracteres y deseamos transmitir este código usando la menor cantidad de bits posible. Muestre cómo representar cualquier código de prefijo óptimo en C usando solo  $2n + 1$  C  $n$   $\text{dlg}$  ne bits. (Sugerencia: use  $2n + 1$  bits para especificar la estructura del árbol, tal como se descubrió al caminar por el árbol).

#### 16.3-7

Generalice el algoritmo de Huffman a palabras clave ternarias (es decir, palabras clave que usan los símbolos 0, 1 y 2) y demuestre que produce códigos ternarios óptimos.

#### 16.3-8

Suponga que un archivo de datos contiene una secuencia de caracteres de 8 bits tal que los 256 caracteres son casi igualmente comunes: la frecuencia máxima de caracteres es menor que el doble de la frecuencia mínima de caracteres. Demuestre que la codificación Huffman en este caso no es más eficiente que usar un código ordinario de longitud fija de 8 bits.

## 16.3-9

Demuestre que ningún esquema de compresión puede esperar comprimir un archivo de caracteres de 8 bits elegidos al azar ni siquiera en un solo bit. (Sugerencia: compare la cantidad de archivos posibles con la cantidad de archivos codificados posibles).

## 16.4 Matroids y métodos codiciosos

En esta sección, esbozamos una hermosa teoría sobre los algoritmos codiciosos. Esta teoría describe muchas situaciones en las que el método codicioso produce soluciones óptimas. Se trata de estructuras combinatorias conocidas como "matroides". Aunque esta teoría no cubre todos los casos en los que se aplica un método codicioso (por ejemplo, no cubre el problema de selección de actividades de la sección 16.1 o el problema de codificación de Huffman de la sección 16.3), sí cubre muchos casos de interés práctico. . Además, esta teoría se ha ampliado para cubrir muchas aplicaciones; consulte las notas al final de este capítulo para obtener referencias.

### matroides

Un matroide es un par ordenado  $M = (S, I)$  cumpliendo las siguientes condiciones.

1.  $S$  es un conjunto finito.
2.  $I$  es una familia no vacía de subconjuntos de  $S$ , llamados subconjuntos independientes de  $S$ , tal que si  $B \in I$  y  $A \subseteq B$ , entonces  $A \in I$ . Decimos que  $I$  es hereditario si cumple esta propiedad. Tenga en cuenta que el conjunto vacío; es necesariamente miembro de  $I$ .
3. Si  $A_1, A_2, \dots, A_k$  y  $|A_j| < |B_j|$ , entonces existe algún elemento  $x \in B_j$  tal que  $A_j \cup \{x\} \in I$ . Decimos que  $I$  satisface la propiedad de cambio.

La palabra "matroid" se debe a Hassler Whitney. Estaba estudiando matroides matriciales, en las que los elementos de  $S$  son las filas de una matriz dada y un conjunto de filas es independiente si son linealmente independientes en el sentido habitual. Como el Ejercicio 16.4-2 le pide que muestre, esta estructura define una matroide.

Como otro ejemplo de matroides, considere el matroid gráfico  $M = (G, I)$  definido en términos de un grafo no dirigido dado  $G = (V, E)$  de la siguiente manera:

El conjunto  $I$  se define como  $E$ , el conjunto de aristas de  $G$ .

Si  $A$  es un subconjunto de  $E$ , entonces  $A \in I$  si y sólo si  $A$  es acíclico. Es decir, un conjunto de aristas  $A$  es independiente si y sólo si el subgrafo  $G[A]$  forma un bosque.

La matroide gráfica  $M$  está estrechamente relacionada con el problema del árbol de expansión mínimo, que se trata en detalle en el Capítulo 23.

**Teorema 16.5**

Si  $G = (V; E)$  es un grafo no dirigido, entonces  $MG(D) \subseteq G$  es un matroide.

Prueba Claramente,  $SG(D)$  es un conjunto finito. Además,  $G$  es hereditario, ya que un subconjunto de un bosque es un bosque. Dicho de otra manera, eliminar bordes de un conjunto acíclico de bordes no puede crear ciclos.

Por lo tanto, queda por demostrar que  $MG$  satisface la propiedad de intercambio. Supongamos que  $GA = (V; A)$  y  $GB = (V; B)$  son bosques de  $G$  y que  $|B| > |A|$ . Es decir,  $A$  y  $B$  son conjuntos acíclicos de aristas y  $B$  contiene más aristas que  $A$ . Decimos que un bosque  $FD = (V; F)$  contiene exactamente  $jVF$  árboles. a los vea por qué, suponga que  $F$  consta de  $t$  árboles, donde el  $i$ -ésimo árbol contiene  $e_i$  vértices aristas. Entonces nosotros tenemos

$$jVF = \sum_{i=1}^t e_i$$

$$D = \sum_{i=1}^t A_i \quad (por el Teorema B.2)$$

$$\begin{aligned} D &= \sum_{i=1}^t A_i \\ &= \sum_{i=1}^t B_i \\ &= jVF \end{aligned}$$

lo que implica que  $t = jVF$ . Por lo tanto, el bosque  $GA$  contiene  $jV - |A|$  árboles, y el bosque  $GB$  contiene  $jV - |B|$  árboles.

Dado que el bosque  $GB$  tiene menos árboles que el bosque  $GA$ , el bosque  $GB$  debe contener algún árbol  $T$  cuyos vértices estén en dos árboles diferentes en el bosque  $GA$ . Además, como  $T$  es conexo, debe contener una arista  $.u; .v$  tal que los vértices  $u$  y  $v$  están en diferentes árboles en el bosque  $GA$ . Desde el borde  $.u; .v$  conecta vértices en dos árboles diferentes en forest  $GA$ , podemos agregar el borde  $.u; .v$  al bosque  $GA$  sin crear un ciclo. Por lo tanto,  $MG$  satisface la propiedad de intercambio, completando la prueba de que  $MG$  es una matroide. ■

Dada una matroide  $M = (S; I)$ , llamamos a un elemento  $x \in S$  una extensión de  $A$  si podemos sumar  $x$  a  $A$  conservando la independencia; es decir,  $x$  es una extensión de  $A$  si  $A \cup \{x\} \in I$ . Como ejemplo, considere una matroide gráfica  $MG$ . Si  $A$  es un conjunto independiente de aristas, entonces la arista  $e$  es una extensión de  $A$  si y solo si  $e$  no está en  $A$  y la adición de  $e$  a  $A$  no crea un ciclo.

Si  $A$  es un subconjunto independiente en una matroide  $M$ , decimos que  $A$  es maximal si no tiene extensiones. Es decir,  $A$  es máximo si no está contenido en ningún subconjunto independiente más grande de  $M$ . La siguiente propiedad suele ser útil.

## Teorema 16.6

Todos los subconjuntos independientes máximos en una matroide tienen el mismo tamaño.

Demostración Supongamos, por el contrario, que  $A$  es un subconjunto independiente máximo de  $M$  y que existe otro subconjunto independiente máximo  $B$  de  $M$ . Entonces, la propiedad de intercambio implica que para algún  $x \in B \setminus A$ , podemos extender  $A$  a un conjunto independiente más grande  $A \cup \{x\}$ , contradiciendo la suposición de que  $A$  es máxima. ■

Como ilustración de este teorema, considere una matroide gráfica  $MG$  para un grafo  $G$  no dirigido y conexo. Cada subconjunto independiente máximo de  $MG$  debe ser un árbol libre con exactamente  $|V| - 1$  aristas que conecte todos los vértices de  $G$ . Tal árbol es llamado árbol de expansión de  $G$ .

Decimos que un matroide  $MD(S)$  es ponderado si está asociado con una función de ponderación  $w$  que asigna un peso estrictamente positivo  $w_x$  a cada elemento  $x \in S$ . La función de ponderación  $w$  se extiende a subconjuntos de  $S$  por suma:

$$w_A = \sum_{x \in A} w_x$$

para cualquier  $A \subseteq S$ . Por ejemplo, si denotamos  $w_e$  el peso de un borde  $e$  en una matroide gráfica  $MG$ , entonces  $w_A$  es el peso total de los bordes en el conjunto de bordes  $A$ .

## Algoritmos codiciosos en un matroide ponderado

Muchos problemas para los que un enfoque codicioso proporciona soluciones óptimas se pueden formular en términos de encontrar un subconjunto independiente de peso máximo en una matroide ponderada. Es decir, se nos da una matroide ponderada  $MD(S)$ , y deseamos encontrar un conjunto independiente  $A$  tal que  $w_A$  esté maximizado. Llamamos a un subconjunto que es independiente y tiene el máximo peso posible un subconjunto óptimo de la matriz. Debido a que el peso  $w_x$  de cualquier elemento  $x \in S$  es positivo, un subconjunto óptimo siempre es un subconjunto maximal independiente; siempre ayuda hacer que  $A$  sea lo más grande posible.

Por ejemplo, en el problema del árbol de expansión mínimo, tenemos un grafo no dirigido  $G$  conexo;  $E$  y una función de longitud  $w$  tal que  $w_e$  es la longitud (positiva) de la arista  $e$ . (Usamos el término "longitud" aquí para referirnos a los pesos de los bordes originales para el gráfico, reservando el término "peso" para referirnos a los pesos en la matroide asociada). Deseamos encontrar un subconjunto de los bordes que conecte todos los vértices juntos y tiene una longitud total mínima. Para ver esto como un problema de encontrar un subconjunto óptimo de un matroide, considere el matroide ponderado  $MG$  con función de peso  $w_0$  donde  $w_0(e) = w_e$  y  $w_0$  es mayor que la longitud máxima de cualquier borde. En esta matroide ponderada, todos los pesos son positivos y un subconjunto óptimo es un árbol de expansión de longitud total mínima en el gráfico original. Más específicamente, cada subconjunto independiente máximo  $A$  corresponde a un árbol de expansión

con  $jV j - 1$  bordes, y desde

$w0 .A/ DX w0 .e/$   
 $e2A$

$DX .w0 nosotros//$   
 $e2A$

$D .jV j 1/w0 X nosotros/$   
 $e2A$

$D .jV j 1/w0 wA/$

para cualquier subconjunto independiente maximal A, un subconjunto independiente que maximiza la cantidad  $w0 .A/$  debe minimizar  $wA/$ . Por lo tanto, cualquier algoritmo que pueda encontrar un subconjunto A óptimo en un matroide arbitrario puede resolver el problema del árbol de expansión mínimo.

El Capítulo 23 proporciona algoritmos para el problema del árbol de expansión mínimo, pero aquí damos un algoritmo codicioso que funciona para cualquier matriz ponderada. El algoritmo toma como entrada una matroide ponderada MD .S; / con una función de peso positiva asociada w, y devuelve un subconjunto óptimo A. En nuestro pseudocódigo, denotamos los componentes de M por M:S y M: y la función de peso por w. El algoritmo es codicioso porque considera a su vez cada elemento x 2 S, en orden de peso monótonamente decreciente, y lo suma inmediatamente al conjunto A siendo acumulado si A [ fxg es independiente.

```
CODICIOSO.M; c/
1 dC;
2 ordenar M:S en orden monótonamente decreciente por peso w 3
para cada x 2 M:S, tomado en orden monótonamente decreciente por peso wx/ if A
4      [ fxg 2 M: ADA
5      [ fxg
6 vuelta A
```

La línea 4 verifica si agregar cada elemento x a A mantendría a A como un conjunto independiente. Si A permanecería independiente, entonces la línea 5 suma x a A. De lo contrario, x se descarta. Dado que el conjunto vacío es independiente, y dado que cada iteración del ciclo for mantiene la independencia de A, el subconjunto A siempre es independiente, por inducción. Por tanto, GREEDY siempre devuelve un subconjunto independiente A. Veremos en un momento que A es un subconjunto de máximo peso posible, por lo que A es un subconjunto óptimo.

El tiempo de ejecución de GREEDY es fácil de analizar. Sea  $n = |S|$ . La fase de clasificación de GREEDY lleva tiempo  $O(n \lg n)$ . La línea 4 se ejecuta exactamente  $n$  veces, una por cada elemento de S. Cada ejecución de la línea 4 requiere una comprobación de si el conjunto A [ fxg es independiente o no. Si cada una de estas comprobaciones toma el tiempo  $O(f(n))$ , todo el algoritmo se ejecuta en el tiempo  $O(n \lg n f(n))$ .

Ahora probamos que GREEDY devuelve un subconjunto óptimo.

**Lema 16.7 (Las matroides exhiben la propiedad de elección codiciosa)**

Supongamos que  $MD .S; /$  es una matroide ponderada con función de peso  $w$  y que  $S$  se ordena monótonamente decreciente por peso. Sea  $x$  el primer elemento de  $S$  tal que  $fxg$  es independiente, si existe tal  $x$ . Si  $x$  existe, entonces existe un subconjunto óptimo  $A$  de  $S$  que contiene  $x$ .

Prueba Si tal  $x$  no existe, entonces el único subconjunto independiente es el conjunto vacío y el lema es vacuamente verdadero. De lo contrario, sea  $B$  cualquier subconjunto óptimo no vacío.

Suponga que  $x \in B$ ; de lo contrario, dejar  $ADB$  da un subconjunto óptimo de  $S$  que contiene  $x$ .

Ningún elemento de  $B$  tiene un peso mayor que  $wx/$ . Para ver por qué, observe que  $y \in B$  implica que  $fyg$  es independiente, ya que  $B$  es hereditario. Nuestra elección de  $x$ , por lo tanto, asegura que  $wx/ \geq wy/$  para cualquier  $y \in B$ .

Construya el conjunto  $A$  de la siguiente manera. Comience con  $AD = fxg$ . Por la elección de  $x$ , el conjunto  $A$  es independiente. Usando la propiedad de intercambio, encuentre repetidamente un nuevo elemento de  $B$  que podamos agregar a  $A$  hasta  $A \cup y \cup B$ , conservando la independencia de  $A$ . En ese punto,  $A$  y  $B$  son iguales excepto que  $A$  tiene  $x$  y  $B$  tiene alguna otra elemento  $y$ .

Es decir,  $ADB \neq fyg$  para algún  $y \in B$ , y así

$wA \geq wB \geq wy \geq wx \geq wB$ :

Como el conjunto  $B$  es óptimo, el conjunto  $A$ , que contiene  $x$ , también debe ser óptimo. ■

A continuación mostramos que si un elemento no es una opción inicialmente, entonces no puede ser una opción más adelante. ■

**Lema 16.8 Sea**

$MD .S; /$  ser cualquier matroide. Si  $x$  es un elemento de  $S$  que es una extensión de algún subconjunto independiente  $A$  de  $S$ , entonces  $x$  también es una extensión de  $A$ .

Prueba Como  $x$  es una extensión de  $A$ , tenemos que  $A[xg]$  es independiente. Como es hereditario,  $fxg$  debe ser independiente. Así,  $x$  es una extensión de  $A$ . ■

**Corolario 16.9**

Sea  $MD .S; /$  ser cualquier matroide. Si  $x$  es un elemento de  $S$  tal que  $x$  no es una extensión de  $A$ , entonces  $x$  no es una extensión de ninguno subconjunto independiente  $A$  de  $S$ .

Prueba Este corolario es simplemente el contrapositivo del Lema 16.8. ■

El corolario 16.9 dice que cualquier elemento que no se puede usar inmediatamente nunca se puede usar. Por lo tanto, GREEDY no puede cometer un error al pasar por alto cualquier elemento inicial en S que no sea una extensión de ;, ya que nunca se pueden usar.

Lema 16.10 (Las matroides exhiben la propiedad de subestructura óptima)

Sea x el primer elemento de S elegido por GREEDY para la matroide ponderada  $M.D.S; /$ . El problema restante de encontrar un subconjunto independiente de peso máximo que contenga x se reduce a encontrar un subconjunto independiente de peso máximo de la matroide ponderada  $M0.D.S0; /$ , donde

$$\begin{aligned} S0 &\subseteq f(y) \cup SW(fx; yg) \\ &\subseteq D(fB(S \setminus fx)) \cup WB(fx; yg) \end{aligned}$$

y la función de peso para  $M0$  es la función de peso para  $M$ , restringida a  $S0$ . (Llamamos  $M0$  a la contracción de  $M$  por el elemento x.)

Prueba Si A es cualquier subconjunto independiente de peso máximo de  $M$  que contiene x, entonces  $A0 = DA \setminus fx$  es un subconjunto independiente de  $M0$ . A la inversa, cualquier subconjunto independiente  $A0$  de  $M0$  produce un subconjunto independiente  $AD = A0 \cup fx$  de  $M$ . Dado que en ambos casos tenemos que  $w(A) = w(A0) + w(x)$ , una solución de peso máximo en  $M$  que contiene x produce una solución de peso máximo en  $M0$ , y viceversa. ■

Teorema 16.11 (Corrección del algoritmo voraz en matroides)

Si  $M.D.S; /$  es un matroide ponderado con función de peso w, luego  $\text{GREEDY}.M; w/$  devuelve un subconjunto óptimo.

Prueba Por el Corolario 16.9, cualquier elemento que GREEDY pasa por alto inicialmente porque no son extensiones de ; pueden ser olvidados, ya que nunca pueden ser útiles. Una vez que GREEDY selecciona el primer elemento x, el Lema 16.7 implica que el algoritmo no se equivoca al sumar x a A, ya que existe un subconjunto óptimo que contiene x. Finalmente, el Lema 16.10 implica que el problema restante es encontrar un subconjunto óptimo en la matroide  $M0$  que sea la contracción de  $M$  por x.

Después de que el procedimiento GREEDY establece A en fxg, podemos interpretar todos sus pasos restantes como si actuaran en la matroide  $M0.D.S0; /$ , porque B es independiente en  $M0$  si y sólo si  $B \setminus fx$  es independiente en  $M$ , para todos los conjuntos  $B \subseteq M$ . Por lo tanto, la operación posterior de GREEDY encontrará un subconjunto independiente de peso máximo para  $M0$  y, la operación general de GREEDY encontrará un subconjunto independiente de peso máximo para  $M$ . ■

## Ejercicios

### 16.4-1

Muestre que  $.S; /$  es un matroide, donde  $S$  es cualquier conjunto finito y  $\mathcal{I}$  es el conjunto de todos los subconjuntos de  $S$  de tamaño máximo  $k$ , donde  $k \leq |S|$ .

### 16.4-2 ?

Dada una matriz  $T$  de  $m \times n$  sobre algún cuerpo (como los reales), demuestre que  $.S; /$  es una matroide, donde  $S$  es el conjunto de columnas de  $T$  y  $A \in \mathcal{I}$  si y sólo si las columnas de  $A$  son linealmente independientes.

### 16.4-3 ?

Demuestre que si  $.S; /$  es un matroide, entonces  $.S^{\circ}; /$  es un matroide, donde

$D \subseteq A^{\circ} \cap S$  contiene un máximo de  $A$  g :

Es decir, los conjuntos independientes máximos de  $.S; /$  son solo los complementos de la conjuntos máximos independientes de  $.S^{\circ}; /$ .

### 16.4-4 ?

Sea  $S$  un conjunto finito y sea  $S_1; S_2; \dots; S_k$  sea una partición de  $S$  en subconjuntos disjuntos no vacíos. Definir la estructura  $.S; /$  por la condición de que  $D \subseteq A \cap S_i$  para  $i = 1; 2; \dots; k$ . Demuestre que  $.S; /$  es un matroide. Es decir, el conjunto de todos los conjuntos  $A$  que contienen como máximo un miembro de cada subconjunto en la partición determina los conjuntos independientes de una matroide.

### 16.4-5

Muestre cómo transformar la función de peso de un problema matroide ponderado, donde la solución óptima deseada es un subconjunto independiente máximo de peso mínimo, para convertirlo en un problema matroide ponderado estándar. Argumente cuidadosamente que su transformación es correcta.

## ? 16.5 Un problema de programación de tareas como matroid

Un problema interesante que podemos resolver usando matroids es el problema de programar de manera óptima tareas de unidad de tiempo en un solo procesador, donde cada tarea tiene una fecha límite, junto con una penalización pagada si la tarea no cumple con su fecha límite. El problema parece complicado, pero podemos resolverlo de una manera sorprendentemente simple convirtiéndolo en un matroide y usando un algoritmo codicioso.

Una tarea de unidad de tiempo es un trabajo, como un programa que se ejecutará en una computadora, que requiere exactamente una unidad de tiempo para completarse. Dado un conjunto finito  $S$  de tareas de unidad de tiempo, un

el horario para S es una permutación de S que especifica el orden en el que se deben realizar estas tareas. La primera tarea de la programación comienza en el momento 0 y finaliza en el momento 1, la segunda tarea comienza en el momento 1 y finaliza en el momento 2, y así sucesivamente.

El problema de programar tareas de unidad de tiempo con plazos y penalizaciones para un solo procesador tiene las siguientes entradas:

un conjunto SD fa1; a2;:::;ang de n tareas por unidad de tiempo; un conjunto de n plazos enteros d1; d2;:::;dn, tal que cada di satisface 1 y se supone que la tarea ai finaliza en el tiempo di; y un conjunto de n pesos o penalizaciones no negativos w1; w2;:::;wn, de modo que incurrimos en una penalización de wi si la tarea ai no finaliza en el tiempo di, y no incurrimos en penalización si una tarea finaliza en su fecha límite.

Deseamos encontrar un programa para S que minimice la penalización total en que se incurre por no cumplir con los plazos.

Consideré un horario dado. Decimos que una tarea está retrasada en este cronograma si finaliza después de su fecha límite. De lo contrario, la tarea es temprana en el cronograma. Siempre podemos transformar un cronograma arbitrario en la forma temprana, en la que las tareas tempranas preceden a las tareas tardías. Para ver por qué, tenga en cuenta que si alguna tarea temprana ai sigue a alguna tarea tardía aj , entonces podemos cambiar las posiciones de ai y aj , y ai seguirá siendo temprana y aj aún estará retrasada.

Además, afirmamos que siempre podemos transformar un cronograma arbitrario en una forma canónica, en la que las tareas tempranas preceden a las tardías y programamos las tareas tempranas en orden de plazos monótonamente crecientes. Para hacerlo, ponemos el cronograma en forma temprana. Luego, siempre que existan dos tareas tempranas ai y aj que finalicen en los respectivos tiempos k y k C 1 en el cronograma de manera que dj < di, intercambiamos las posiciones de ai y aj . Dado que aj es anterior al intercambio, k C 1 dj .

Por lo tanto, k C 1<di , por lo que ai todavía es temprano después del intercambio. Debido a que la tarea aj se mueve antes en el cronograma, permanece temprano después del intercambio.

La búsqueda de un cronograma óptimo se reduce así a encontrar un conjunto A de tareas que asignamos al inicio del cronograma óptimo. Habiendo determinado A, podemos crear el cronograma real enumerando los elementos de A en orden de plazos monótonamente crecientes, luego enumerando las tareas atrasadas (es decir, SA) en cualquier orden, produciendo un ordenamiento canónico del cronograma óptimo.

Decimos que un conjunto A de tareas es independiente si existe un cronograma para estas tareas tal que ninguna se atrasa. Claramente, el conjunto de tareas tempranas para un cronograma forma un conjunto independiente de tareas. Denotemos el conjunto de todos los conjuntos independientes de tareas.

Consideré el problema de determinar si un conjunto A dado de tareas es independiente. Para t D 0; 1; 2; : : : ; n, sea Nt.A/ el número de tareas en A cuya fecha límite es t o anterior. Tenga en cuenta que N0.A/ D 0 para cualquier conjunto A.

**Lema 16.12 Para**

cualquier conjunto de tareas A, las siguientes declaraciones son equivalentes.

1. El conjunto A es independiente.
2. Para  $t \in 0; 1; 2; \dots; n$ , tenemos  $Nt.A / t$ .
3. Si las tareas en A están programadas en orden de plazos crecientes monótonamente, entonces ninguna tarea llega tarde.

Prueba Para mostrar que (1) implica (2), probamos la contrapositiva: si  $Nt.A / t$  para algún  $t$ , entonces no hay manera de hacer un horario sin tareas atrasadas para el conjunto A, porque más de  $t$  tareas debe terminar antes del tiempo  $t$ . Por lo tanto, (1) implica (2). Si (2) se cumple, entonces (3) debe seguir: no hay forma de "atascarse" al programar las tareas en orden de plazos crecientes monótonamente, ya que (2) implica que el  $i$ -ésimo plazo más grande es al menos  $i$ . Finalmente, (3) implica trivialmente (1). ■

Usando la propiedad 2 del Lema 16.12, podemos calcular fácilmente si un determinado conjunto de tareas es independiente (ver Ejercicio 16.5-2).

El problema de minimizar la suma de las penalizaciones de las tareas tardías es el mismo que el problema de maximizar la suma de las penalizaciones de las primeras tareas. El siguiente teorema asegura que podemos usar el algoritmo voraz para encontrar un conjunto independiente A de tareas con la máxima penalización total.

**Teorema 16.13 Si**

S es un conjunto de tareas por unidad de tiempo con plazos, y es el conjunto de todos los conjuntos de tareas independientes, entonces el sistema correspondiente  $.S /$  es un matroide.

Prueba Cada subconjunto de un conjunto independiente de tareas es ciertamente independiente. Para probar la propiedad de intercambio, suponga que B y A son conjuntos independientes de tareas y que  $jBj > jAj$ . Sea  $k$  el mayor  $t$  tal que  $Nt.B / Nt.A /$ . (Tal valor de  $t$  existe, ya que  $N0.A / D N0.B / D 0$ ) Dado que  $Nn.B / D jBj & Nn.A / D jAj$ , pero  $jBj > jAj$ , debemos tener que  $k < n$  y que  $Nj.B / > Nj.A /$  para todo  $j$  en el rango  $k \leq j \leq n$ . Por lo tanto, B contiene más tareas con fecha límite  $k$  que A. Sea  $a_i$  una tarea en BA con fecha límite  $k$ . Sea  $A0 = DA[a_i]$ .

Ahora mostramos que  $A0$  debe ser independiente usando la propiedad 2 del Lema 16.12. Para  $t \in k$  tenemos  $Nt.A0 / D Nt.A / t$ , ya que A es independiente. Para  $t < k$ , tenemos  $Nt.A0 / Nt.B / t$ , ya que B es independiente. Por tanto,  $A0$  es independiente, completando nuestra prueba de que  $.S /$  es un matroide. ■

Por el teorema 16.11, podemos usar un algoritmo voraz para encontrar un conjunto independiente de tareas A de peso máximo. Luego podemos crear un cronograma óptimo que tenga las tareas en A como sus primeras tareas. Este método es un algoritmo eficiente para programar

Tarea						
	70	60	50	40	30	20
W <sub>1</sub> a <sub>1</sub>	1	0	0	0	0	0
W <sub>2</sub> a <sub>2</sub>	0	1	0	0	0	0
W <sub>3</sub> a <sub>3</sub>	0	0	1	0	0	0
W <sub>4</sub> a <sub>4</sub>	0	0	0	1	0	0
W <sub>5</sub> a <sub>5</sub>	0	0	0	0	1	0
W <sub>6</sub> a <sub>6</sub>	0	0	0	0	0	1
W <sub>7</sub> a <sub>7</sub>	0	0	0	0	0	0

Figura 16.7 Una instancia del problema de programar tareas de unidad de tiempo con plazos y penalizaciones para un solo procesador.

tareas de unidad de tiempo con plazos y penalizaciones para un solo procesador. El tiempo de ejecución es  $O(n^2)$  usando CODICIOSO, ya que cada una de las comprobaciones de independencia  $O(n)$  realizadas por ese algoritmo toma tiempo  $O(n)$  (vea el Ejercicio 16.5-2). El problema 16-4 da una implementación más rápida.

La figura 16.7 muestra un ejemplo del problema de programar tareas de unidad de tiempo con plazos y penalizaciones para un solo procesador. En este ejemplo, el algoritmo voraz selecciona, en orden, las tareas  $a_1, a_2, a_3$  y  $a_4$ , luego rechaza  $a_5$  (porque  $N_4.f(a_1; a_2; a_3; a_4; a_5) > D_5$ ) y  $a_6$  (porque  $N_4.f(a_1; a_2; a_3; a_4; a_6) > D_5$ ), y finalmente acepta  $a_7$ . El programa óptimo final es

$ha_2; a_4; a_1; a_3; a_7; a_5; a_6;$

que tiene una penalización total incurrida de  $w_5 C w_6 D 50$ .

### Ejercicios

#### 16.5-1

Resuelva la instancia del problema de programación que se muestra en la figura 16.7, pero con cada penalización  $w_i$  reemplazada por  $80 w_i$ .

#### 16.5-2

Muestre cómo usar la propiedad 2 del Lema 16.12 para determinar en el tiempo  $O(jA_j)$  si un conjunto  $A$  dado de tareas es o no independiente.

## Problemas

### 16-1 Cambio de monedas

Considere el problema de cambiar  $n$  centavos usando la menor cantidad de monedas. Suponga que el valor de cada moneda es un número entero.

- Describa un algoritmo codicioso para dar cambio que consista en monedas de veinticinco centavos, diez centavos, cinco centavos y centavos. Demuestre que su algoritmo produce una solución óptima.

b. Supongamos que las monedas disponibles están en las denominaciones que son potencias de c, es decir, las denominaciones son  $c^0; c^1; \dots; c^k$  para algunos enteros  $c > 1$  y  $k \geq 1$ .

Demuestre que el algoritmo codicioso siempre produce una solución óptima.

C. Proporcione un conjunto de denominaciones de monedas para las que el algoritmo voraz no produzca una solución óptima. Su conjunto debe incluir un centavo para que haya una solución para cada valor de  $n$ .

d. Proporcione un algoritmo  $O(nk)$ -time que dé cambio para cualquier conjunto de  $k$  denominaciones de monedas diferentes, suponiendo que una de las monedas sea un centavo.

#### 16-2 Programación para minimizar el tiempo promedio de finalización

Suponga que le dan un SD f(a<sub>1</sub>, a<sub>2</sub>, ..., a<sub>n</sub>) de tareas, donde la tarea a<sub>i</sub> requiere unidades p<sub>i</sub> de tiempo de procesamiento para completarse, una vez que ha comenzado. Tiene una computadora en la que ejecutar estas tareas, y la computadora solo puede ejecutar una tarea a la vez. Sea ci el tiempo de finalización de la tarea a<sub>i</sub>, es decir, el tiempo en el que la tarea a<sub>i</sub> completa el procesamiento. Su objetivo es minimizar el tiempo promedio de finalización, es decir, minimizar  $\frac{1}{n} \sum_i p_i c_i$ . Por ejemplo, suponga que hay dos tareas a<sub>1</sub> y a<sub>2</sub>, con p<sub>1</sub> = 3 y p<sub>2</sub> = 5, y considere el programa en el que a<sub>2</sub> se ejecuta primero, seguido de a<sub>1</sub>. Entonces c<sub>2</sub> = 5, c<sub>1</sub> = 8, y el tiempo promedio de terminación es  $\frac{1}{2}(5 + 8) = 6.5$ .

Sin embargo, si la tarea a<sub>1</sub> se ejecuta primero, entonces c<sub>1</sub> = 3, c<sub>2</sub> = 8, y el tiempo promedio de finalización es  $\frac{1}{2}(3 + 8) = 5.5$ .

a. Proporcione un algoritmo que programe las tareas para minimizar el tiempo promedio de finalización.

Cada tarea debe ejecutarse de forma no preventiva, es decir, una vez que se inicia la tarea a<sub>i</sub>, debe ejecutarse continuamente durante p<sub>i</sub> unidades de tiempo. Demuestre que su algoritmo minimiza el tiempo promedio de finalización y establezca el tiempo de ejecución de su algoritmo.

b. Supongamos ahora que las tareas no están todas disponibles a la vez. Es decir, cada tarea no puede comenzar hasta su hora de liberación r<sub>i</sub>. Supongamos también que permitimos la preferencia, de modo que una tarea pueda suspenderse y reiniciarse en un momento posterior. Por ejemplo, una tarea a<sub>i</sub> con un tiempo de procesamiento p<sub>i</sub> = 6 y un tiempo de liberación r<sub>i</sub> = 1 podría comenzar a ejecutarse en el tiempo 1 y ser reemplazada en el tiempo 4. Luego podría reanudarse en el tiempo 10 pero ser reemplazada en el tiempo 11 y finalmente podría reanudarse en el tiempo 13 y completarse en el tiempo 15. La tarea a<sub>i</sub> se ejecutó durante un total de 6 unidades de tiempo, pero su tiempo de ejecución se dividió en tres partes. En este escenario, el tiempo de finalización de a<sub>i</sub> es 15.

Proporcione un algoritmo que programe las tareas para minimizar el tiempo promedio de finalización en este nuevo escenario. Demuestre que su algoritmo minimiza el tiempo promedio de finalización y establezca el tiempo de ejecución de su algoritmo.

## 16-3 Subgrafos acíclicos a. La

matriz de incidencia para un grafo no dirigido  $G = (V; E)$  es una matriz  $M : V \times E$  tal que  $M_{v,e} = 1$  si la arista  $e$  incide sobre el vértice  $v$  y  $M_{v,e} = 0$  en caso contrario. Argumente qué un conjunto de columnas de  $M$  es linealmente independiente sobre el campo de los enteros módulo 2 si y solo si el conjunto de aristas correspondiente es acíclico.

Luego, use el resultado del ejercicio 16.4-2 para proporcionar una prueba alternativa de que  $(E; /)$  de la parte (a) es un matroide.

b. Supongamos que asociamos un peso no negativo  $w_e$  con cada arista en un grafo no dirigido  $G = (V; E)$ . Proporcione un algoritmo eficiente para encontrar un subconjunto acíclico de  $E$  de peso total máximo.

C. Sea  $G = (V; E)$  un grafo dirigido arbitrario, y sea  $(E; /)$  definirse de modo que  $A \in 2^E$  si y sólo si  $A$  no contiene ningún ciclo dirigido. Dé un ejemplo de un grafo dirigido  $G$  tal que el sistema asociado  $(E; /)$  no es un matroide.

Especifique qué condición definitoria para un matroide no se cumple.

d. La matriz de incidencia para un grafo dirigido  $G = (V; E)$  sin bucles propios es una matriz  $M : V \times E$  tal que  $M_{v,e} = 1$  si la arista  $e$  sale del vértice  $v$ ,  $M_{v,e} = -1$  si la arista  $e$  entra en el vértice  $v$  y  $M_{v,e} = 0$  en caso contrario. Argumente que si un conjunto de columnas de  $M$  es linealmente independiente, entonces el correspondiente conjunto de aristas no contiene un ciclo dirigido.

mi. El ejercicio 16.4-2 nos dice que el conjunto de conjuntos de columnas linealmente independientes de cualquier matriz  $M$  forma un matroide. Explique cuidadosamente por qué los resultados de las partes (d) y (e) no son contradictorios. ¿Cómo puede no haber una correspondencia perfecta entre la noción de que un conjunto de aristas es acíclico y la noción de que el conjunto asociado de columnas de la matriz de incidencia es linealmente independiente?

## 16-4 Variaciones de programación

Considere el siguiente algoritmo para el problema de la sección 16.5 de programación de tareas de unidad de tiempo con fechas límite y penalizaciones. Deje que todos los  $n$  intervalos de tiempo estén inicialmente vacíos, donde el intervalo de tiempo  $i$  es el intervalo de tiempo de longitud unitaria que termina en el tiempo  $i$ . Consideraremos las tareas en orden de penalización monótonamente decreciente. Al considerar la tarea  $a_j$ , si existe un intervalo de tiempo en  $o$  antes de la fecha límite de  $a_j$  que aún está vacío, asigne  $a_j$  al último intervalo de ese tipo y llénelo. Si no existe tal ranura, asigne la tarea  $a_j$  a la última de las ranuras aún sin llenar.

a. Argumente que este algoritmo siempre da una respuesta óptima.

b. Use el bosque rápido de conjuntos disjuntos presentado en la Sección 21.3 para implementar el algoritmo de manera eficiente. Suponga que el conjunto de tareas de entrada ya se ha clasificado en

orden monótonamente decreciente por penalización. Analice el tiempo de ejecución de su implementación.

#### 16-5 Almacenamiento en caché fuera de línea

Las computadoras modernas usan un caché para almacenar una pequeña cantidad de datos en una memoria rápida. Aunque un programa puede acceder a grandes cantidades de datos, al almacenar un pequeño subconjunto de la memoria principal en la memoria caché (una memoria pequeña pero más rápida), el tiempo total de acceso puede disminuir considerablemente. Cuando se ejecuta un programa de computadora, hace una secuencia  $hr_1; r_2; \dots; r_m$  de n solicitudes de memoria, donde cada solicitud es para un elemento de datos en particular. Por ejemplo, un programa que accede a 4 elementos distintos  $f_a; b; C; dg$  podría hacer la secuencia de solicitudes  $hd; b; d; b; d; a; C; d; b; a; C; bi$ . Sea  $k$  el tamaño del caché. Cuando la caché contiene  $k$  elementos y el programa solicita el elemento  $r_i$ , el sistema debe decidir, para esta y cada solicitud posterior, qué  $k$  elementos conservar en la caché. Más precisamente, para cada solicitud  $r_i$ , el algoritmo de gestión de caché verifica si el elemento  $r_i$  ya está en el caché. Si es así, entonces tenemos un acierto de caché; de lo contrario, tenemos una falta de caché. Ante un error de caché, el sistema recupera  $r_i$  de la memoria principal y el algoritmo de administración de caché debe decidir si mantener  $r_i$  en el caché. Si decide mantener  $r_i$  y el caché ya contiene  $k$  elementos, entonces debe desalojar un elemento para hacer espacio para  $r_i$ . El algoritmo de administración de caché expulsa datos con el objetivo de minimizar la cantidad de errores de caché en toda la secuencia de solicitudes.

Por lo general, el almacenamiento en caché es un problema en línea. Es decir, tenemos que tomar decisiones sobre qué datos guardar en la caché sin conocer las futuras solicitudes. Aquí, sin embargo, consideramos la versión fuera de línea de este problema, en la que se nos da de antemano la secuencia completa de  $n$  solicitudes y el tamaño de caché  $k$ , y deseamos minimizar el número total de errores de caché.

Podemos resolver este problema fuera de línea mediante una estrategia codiciosa llamada más lejano en el futuro, que elige desalojar el elemento en el caché cuyo próximo acceso en la secuencia de solicitud llega más lejos en el futuro.

a. Escriba un pseudocódigo para un administrador de caché que utilice la estrategia del futuro más lejano.

La entrada debe ser una secuencia  $hr_1; r_2; \dots; r_m$  de solicitudes y un tamaño de caché  $k$ , y el resultado debe ser una secuencia de decisiones sobre qué elemento de datos (si corresponde) desalojar en cada solicitud. ¿Cuál es el tiempo de ejecución de su algoritmo?

b. Muestre que el problema de almacenamiento en caché fuera de línea exhibe una subestructura óptima.

C. Demostrar que más lejano en el futuro produce el mínimo número posible de caché extraña

### Notas del capítulo

Se puede encontrar mucho más material sobre algoritmos codiciosos y matroides en Lawler [224] y Papadimitriou y Steiglitz [271].

El algoritmo codicioso apareció por primera vez en la literatura de optimización combinatoria en un artículo de 1971 de Edmonds [101], aunque la teoría de las matroides se remonta a un artículo de 1935 de Whitney [355].

Nuestra prueba de la corrección del algoritmo voraz para el problema de selección de actividad se basa en la de Gavril [131]. El problema de la programación de tareas se estudia en Lawler [224]; Horowitz, Sahni y Rajasekaran [181]; y Brassard y Bratley [54].

Los códigos Huffman se inventaron en 1952 [185]; Lelewer y Hirschberg [231] survey técnicas de compresión de datos conocidas a partir de 1987.

Korte y Lov'asz [216, 217, 218, 219], quienes generalizan mucho la teoría aquí presentada.

---

## 17 Análisis amortizado

En un análisis amortizado, promediamos el tiempo requerido para realizar una secuencia de operaciones de estructura de datos sobre todas las operaciones realizadas. Con el análisis amortizado, podemos mostrar que el costo promedio de una operación es pequeño, si promediamos sobre una secuencia de operaciones, aunque una sola operación dentro de la secuencia pueda ser costosa. El análisis amortizado difiere del análisis del caso promedio en que la probabilidad no está involucrada; un análisis amortizado garantiza el desempeño promedio de cada operación en el peor de los casos.

Las primeras tres secciones de este capítulo cubren las tres técnicas más comunes utilizadas en el análisis amortizado. La sección 17.1 comienza con un análisis agregado, en el que determinamos un límite superior  $T \cdot n$  sobre el costo total de una secuencia de  $n$  operaciones. El costo promedio por operación es entonces  $T \cdot n/n$ . Tomamos el coste medio como coste amortizado de cada operación, de manera que todas las operaciones tienen el mismo coste amortizado.

La Sección 17.2 cubre el método contable, en el cual determinamos un costo amortizado de cada operación. Cuando exista más de un tipo de operación, cada tipo de operación podrá tener un costo amortizado diferente. El método de contabilidad sobrecarga algunas operaciones al principio de la secuencia, almacenando el sobrecargo como "crédito prepago" en objetos específicos en la estructura de datos. Más adelante en la secuencia, el crédito paga las operaciones que se cobran menos de lo que realmente cuestan.

La sección 17.3 analiza el método potencial, que es como el método contable en el sentido de que determinamos el costo amortizado de cada operación y es posible que cobremos de más las operaciones al principio para compensar los cargos deficientes más adelante. El método potencial mantiene el crédito como la "energía potencial" de la estructura de datos como un todo en lugar de asociar el crédito con objetos individuales dentro de la estructura de datos.

Usaremos dos ejemplos para examinar estos tres métodos. Una es una pila con la operación adicional MULTIPOP, que extrae varios objetos a la vez. El otro es un contador binario que cuenta progresivamente desde 0 mediante la operación única INCREMENTO.

Al leer este capítulo, tenga en cuenta que los cargos asignados durante un análisis amortizado son solo para fines de análisis. No necesitan, y no deberían, aparecer en el código. Si, por ejemplo, asignamos un crédito a un objeto x cuando usamos el método de contabilidad, no tenemos necesidad de asignar una cantidad apropiada a algún atributo, como x: crédito, en el código.

Cuando realizamos un análisis amortizado, a menudo obtenemos información sobre una estructura de datos en particular, y esta información puede ayudarnos a optimizar el diseño. En la sección 17.4, por ejemplo, usaremos el método potencial para analizar una mesa que se expande y se contrae dinámicamente.

---

## 17.1 Análisis agregado

En el análisis agregado, mostramos que para todo n, una secuencia de n operaciones toma el tiempo  $T \cdot n$  en el peor de los casos en total. En el peor de los casos, el coste medio, o coste amortizado, por operación es por tanto  $T \cdot n/n = n$ . Tenga en cuenta que este costo amortizado se aplica a cada operación, incluso cuando hay varios tipos de operaciones en la secuencia.

Los otros dos métodos que estudiaremos en este capítulo, el método contable y el método potencial, pueden asignar diferentes costos amortizados a diferentes tipos de operaciones.

operaciones de pila

En nuestro primer ejemplo de análisis agregado, analizamos pilas que se han aumentado con una nueva operación. La Sección 10.1 presentó las dos operaciones fundamentales de la pila, cada una de las cuales toma  $O(1)$  tiempo:

PUSH.S; x/ empuja el objeto x a la pila S.

POP.S / extrae la parte superior de la pila S y devuelve el objeto extraído. Llamar a POP en una pila vacía genera un error.

Dado que cada una de estas operaciones se ejecuta en  $O(1)$  tiempo, consideraremos que el costo de cada una es 1. El costo total de una secuencia de n operaciones PUSH y POP es, por lo tanto, n, y el tiempo de ejecución real para n operaciones es, por lo tanto,  $n \cdot O(1) = n$ .

Ahora agregamos la operación de pila MULTIPOP.S; k/, que elimina los k objetos superiores de la pila S, extrayendo toda la pila si la pila contiene menos de k objetos.

Por supuesto, suponemos que k es positivo; de lo contrario, la operación MULTIPOP deja la pila sin cambios. En el siguiente pseudocódigo, la operación STACK-EMPTY devuelve VERDADERO si no hay objetos actualmente en la pila y FALSO en caso contrario.

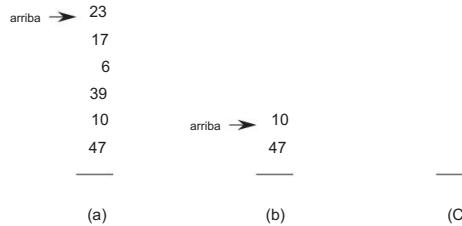


Figura 17.1 La acción de MULTIPOP en una pila S, mostrada inicialmente en (a). Los 4 objetos superiores son extraídos por  $\text{MULTIPOP.S; } 4/$ , cuyo resultado se muestra en (b). La siguiente operación es  $\text{MULTIPOP.S; } 7/$ , que vacía la pila, que se muestra en (c), ya que quedaban menos de 7 objetos.

$\text{MULTIPOP.S; } k/ 1$

mientras no  $\text{STACK-EMPTY.S} / y \ k > 0 \ 2 \ \text{POP.S} / k$

D k 1  
3

La Figura 17.1 muestra un ejemplo de MULTIPOP.

¿Cuál es el tiempo de ejecución de  $\text{MULTIPOP.S; } k/$  en una pila de s objetos? El tiempo de ejecución real es lineal en el número de operaciones POP realmente ejecutadas y, por lo tanto, podemos analizar MULTIPOP en términos de costos abstractos de 1 para PUSH y POP. El número de iteraciones del ciclo while es el número  $\min.s; k/$  de objetos saltados de la pila. Cada iteración del ciclo realiza una llamada a POP en la línea 2. Por lo tanto, el costo total de MULTIPOP es  $\min.s; k/$ , y el tiempo de ejecución real es una función lineal de este costo.

Analicemos una secuencia de n operaciones PUSH, POP y MULTIPOP en una pila inicialmente vacía. El costo en el peor de los casos de una operación MULTIPOP en la secuencia es  $O(n)$ , ya que el tamaño de la pila es como máximo n. Por lo tanto, el tiempo en el peor de los casos de cualquier operación de pila es  $O(n)$  y, por lo tanto, una secuencia de n operaciones cuesta  $O.n^2/$ , ya que podemos tener operaciones  $O(1)$  MULTIPOP que cuestan  $O(1)$  cada una. Aunque este análisis es correcto, el resultado  $O.n^2/$ , que obtuvimos al considerar individualmente el costo del peor de los casos de cada operación, no es ajustado.

Usando el análisis agregado, podemos obtener un mejor límite superior que considere la secuencia completa de n operaciones. De hecho, aunque una sola operación MULTIPOP puede ser costosa, cualquier secuencia de n operaciones PUSH, POP y MULTIPOP en una pila inicialmente vacía puede costar como máximo  $O(n)$ . ¿Por qué? Podemos sacar cada objeto de la pila como máximo una vez por cada vez que lo empujamos a la pila. Por lo tanto, la cantidad de veces que se puede llamar a POP en una pila no vacía, incluidas las llamadas dentro de MULTIPOP, es como máximo la cantidad de operaciones PUSH, que es como máximo n. Para cualquier valor de n, cualquier secuencia de n operaciones PUSH, POP y MULTIPOP toma un total de  $O(n)$  time. El coste medio de una operación es  $O(1)$ . Además

análisis, asignamos el costo amortizado de cada operación como el costo promedio. En este ejemplo, por lo tanto, las tres operaciones de apilamiento tienen un costo amortizado de  $O.1/$ .

Hacemos hincapié nuevamente en que aunque acabamos de mostrar que el costo promedio y, por lo tanto, el tiempo de ejecución de una operación de pila es  $O.1/$ , no usamos un razonamiento probabilístico. De hecho, mostramos un límite en el peor de los casos de On/ en una secuencia de n operaciones. Dividiendo este costo total por n se obtuvo el costo promedio por operación, o el costo amortizado.

#### Incrementando un contador binario

Como otro ejemplo de análisis agregado, considere el problema de implementar un contador binario de k bits que cuenta hacia arriba desde 0. Usamos una matriz AŒ0 : : k 1 de bits, donde A:longitud k , como contador. Un número binario x que se almacena en el contador tiene su bit de orden más bajo en AŒ0 y su bit de orden más alto en AŒk 1, de modo que  $x \equiv P_k AŒi 2^i$ . Inicialmente,  $x \equiv 0$ , y por lo tanto AŒi D 0 para  $i=0; 1; \dots; k-1$ . Para sumar 1 (módulo  $2^k$ ) al valor del contador, usamos el siguiente procedimiento.

```
INCREMENT.A/ 1 i
D 0 2 while
i < A:longitud y AŒi == 1 3 AŒi D 0 4 i D i C 1
```

5 si  $i < A:longitud$  6

AŒi D 1

La figura 17.2 muestra lo que sucede con un contador binario cuando lo incrementamos 16 veces, comenzando con el valor inicial 0 y terminando con el valor 16. Al comienzo de cada iteración del ciclo while en las líneas 2 a 4, deseamos agregar un 1 en la posición i.

Si AŒi D 1, luego agregar 1 cambia el bit a 0 en la posición i y produce un acarreo de 1, que se agregará a la posición i C 1 en la siguiente iteración del bucle. De lo contrario, el ciclo termina, y luego, si  $i < k$ , sabemos que AŒi D 0, por lo que la línea 6 suma un 1 en la posición i, convirtiendo el 0 en 1. El costo de cada operación de INCREMENTO es lineal en el número de bits volteados.

Al igual que con el ejemplo de la pila, un análisis superficial produce un límite que es correcto pero no estricto. Una sola ejecución de INCREMENT toma un tiempo  $.k/$  en el peor de los casos, en el que la matriz A contiene todos los 1. Así, una secuencia de n operaciones de INCREMENTO en un contador inicialmente cero toma tiempo  $O.nk/$  en el peor de los casos.

Podemos ajustar nuestro análisis para producir un costo de On/ en el peor de los casos para una secuencia de n operaciones de INCREMENTO al observar que no todos los bits cambian cada vez que se llama a INCREMENTO . Como muestra la figura 17.2, AŒ0 cambia cada vez que se llama a INCREMENT . El siguiente bit hacia arriba, AŒ1, cambia solo cada dos veces: una secuencia de n INCREMENTO

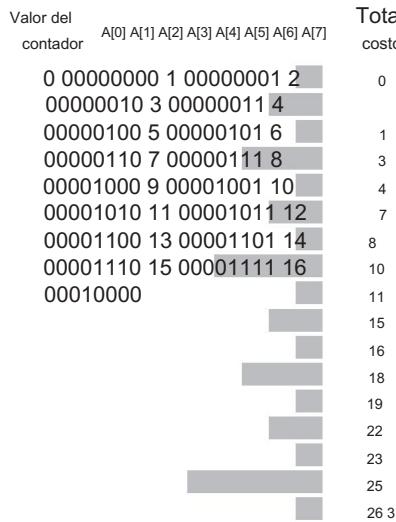


Figura 17.2 Un contador binario de 8 bits cuyo valor va de 0 a 16 mediante una secuencia de 16 operaciones de INCREMENTO . Los bits que cambian para alcanzar el siguiente valor están sombreados. El costo de funcionamiento para voltear bits se muestra a la derecha. Observe que el costo total siempre es menor que el doble del número total de operaciones INCREMENTO .

las operaciones en un contador inicialmente cero hacen que AOE1 cambie  $bn=2c$  veces. De manera similar, el bit AOE2 voltea solo cada cuarta vez, o  $bn=4c$  veces en una secuencia de  $n$  veces en una operación. En general, para  $i \in 0; \dots; k-1$ , el bit AOE $i$  voltea  $bn=2^i c$  veces en una secuencia de  $n$  operaciones de INCREMENTO en un contador inicialmente cero. Para  $i=k$ , el bit AOE $i$  no existe, por lo que no puede cambiar. El número total de vueltas en la secuencia es, por lo tanto,

$$\sum_{i=0}^{k-1} \frac{n}{2^i}$$

por la ecuación (A.6). Por lo tanto, el tiempo en el peor de los casos para una secuencia de  $n$  operaciones de INCREMENTO en un contador inicialmente cero es  $On$ . El coste medio de cada operación, y por tanto el coste amortizado por operación, es  $On/n = D O.1$ .

## Ejercicios

## 17.1-1

Si el conjunto de operaciones de la pila incluyera una operación MULTIPUSH , que empuja k artículos a la pila, ¿se mantendría el límite O.1/ del costo amortizado de las operaciones de la pila?

## 17.1-2

Muestre que si se incluyera una operación DECREMENT en el ejemplo del contador de k bits, n operaciones podrían costar tanto como  $,nk/\text{tiempo}$ .

## 17.1-3

Supongamos que realizamos una secuencia de n operaciones en una estructura de datos en la que la i-ésima operación cuesta  $i$  si  $i$  es una potencia exacta de 2, y 1 en caso contrario. Utilice el análisis agregado para determinar el costo amortizado por operación.

## 17.2 El método contable

En el método contable de análisis amortizado, asignamos diferentes cargos a diferentes operaciones, con algunas operaciones cargadas más o menos de lo que realmente cuestan. A la cantidad que cobramos de una operación la llamamos coste amortizado. Cuando el costo amortizado de una operación excede su costo real, asignamos la diferencia a objetos específicos en la estructura de datos como crédito. El crédito puede ayudar a pagar operaciones posteriores cuyo costo amortizado es menor que su costo real. Por lo tanto, podemos ver el costo amortizado de una operación dividido entre su costo real y el crédito que se deposita o se agota. Diferentes operaciones pueden tener diferentes costos amortizados. Este método difiere del análisis agregado, en el que todas las operaciones tienen el mismo costo amortizado.

Debemos elegir cuidadosamente los costos amortizados de las operaciones. Si queremos mostrar que, en el peor de los casos, el costo promedio por operación es pequeño mediante el análisis con costos amortizados, debemos asegurarnos de que el costo total amortizado de una secuencia de operaciones proporcione un límite superior del costo real total de la secuencia. Además, como en el análisis agregado, esta relación debe mantenerse para todas las secuencias de operaciones. Si denotamos el costo real de la i-ésima operación por  $c_i$  y el costo amortizado de la i-ésima operación por  $c_{yi}$ , requerimos

$$\sum_{i=1}^n c_{yi} \leq \sum_{i=1}^n c_i \quad (17.1)$$

para todas las secuencias de n operaciones. El crédito total almacenado en la estructura de datos es la diferencia entre el costo total amortizado y el costo real total, o

Por desigualdad (17.1), el crédito total asociado con la estructura de datos debe ser no negativo en todo momento. Si alguna vez permitiéramos que el crédito total se volviera negativo (el resultado de cargar de menos operaciones tempranas con la promesa de reembolsar la cuenta más adelante), entonces los costos amortizados totales incurridos en ese momento estarían por debajo de los costos reales totales incurridos; para la secuencia de operaciones hasta ese momento, el costo total amortizado no sería un límite superior del costo real total. Por lo tanto, debemos cuidar que el crédito total en la estructura de datos nunca sea negativo.

### operaciones de pila

Para ilustrar el método contable del análisis amortizado, volvamos al ejemplo de la pila. Recuerde que los costos reales de las operaciones fueron

EMPUJAR	1 ,
ESTALLIDO	1 ,
MULTIPOP min.k; s/ ,	

donde k es el argumento proporcionado a MULTIPOP y s es el tamaño de la pila cuando se llama. Asignemos los siguientes costos amortizados:

EMPUJAR	2 ,
ESTALLIDO	0 ,
MULTIPOP 0 .	

Nótese que el costo amortizado de MULTIPOP es constante (0), mientras que el costo real es variable. Aquí, los tres costos amortizados son constantes. En general, los costos amortizados de las operaciones consideradas pueden diferir entre sí, e incluso pueden diferir asintóticamente.

Ahora mostraremos que podemos pagar cualquier secuencia de operaciones de pila cargando los costos amortizados. Supongamos que usamos un billete de dólar para representar cada unidad de costo. Comenzamos con una pila vacía. Recuerde la analogía de la Sección 10.1 entre la estructura de datos de pila y una pila de platos en una cafetería. Cuando empujamos un plato en la pila, usamos 1 dólar para pagar el costo real del empuje y nos queda un crédito de 1 dólar (de los 2 dólares cargados), que dejamos encima del plato. En cualquier momento, cada plato de la pila tiene un dólar de crédito.

El dólar almacenado en el plato sirve como prepago por el costo de sacarlo de la pila. Cuando ejecutamos una operación POP , no cobramos nada por la operación y pagamos su costo real utilizando el crédito almacenado en la pila. Para abrir un plato, quitamos el dólar de crédito del plato y lo usamos para pagar el costo real de la operación. Así, cargando un poco más la operación PUSH , no podemos cargar nada la operación POP .

Además, también podemos cobrar nada las operaciones de MULTIPOP . Para abrir el primer plato, sacamos el dólar de crédito del plato y lo usamos para pagar el costo real de una operación POP . Para abrir un segundo plato, nuevamente tenemos un dólar de crédito en el plato para pagar la operación POP , y así sucesivamente. Por eso, siempre hemos cobrado lo suficiente por adelantado para pagar las operaciones de MULTIPOP . En otras palabras, dado que cada plato de la pila tiene 1 dólar de crédito y la pila siempre tiene un número no negativo de platos, nos hemos asegurado de que la cantidad de crédito siempre sea no negativa. Por lo tanto, para cualquier secuencia de  $n$  operaciones PUSH, POP y MULTIPOP , el costo total amortizado es un límite superior del costo total real. Dado que el costo total amortizado es On/, también lo es el costo real total.

### Incrementando un contador binario

Como otra ilustración del método contable, analizamos la operación INCREMENTO en un contador binario que comienza en cero. Como observamos anteriormente, el tiempo de ejecución de esta operación es proporcional a la cantidad de bits invertidos, que usaremos como nuestro costo para este ejemplo. Una vez más, usemos un billete de un dólar para representar cada unidad de costo (la inversión de un bit en este ejemplo).

Para el análisis amortizado, carguemos un costo amortizado de 2 dólares para configurar un bit en 1. Cuando se configura un bit, usamos 1 dólar (de los 2 dólares cobrados) para pagar la configuración real del bit, y colocamos el otro dólar en el bit como crédito para usarlo más tarde cuando volvamos a poner el bit en 0. En cualquier momento, cada 1 en el contador tiene un dólar de crédito y, por lo tanto, no podemos cobrar nada para restablecer un poco a 0; solo pagamos el reinicio con el billete de un dólar en la broca.

Ahora podemos determinar el costo amortizado de INCREMENTO. El costo de restablecer los bits dentro del bucle while se paga con los dólares de los bits que se restablecen. El procedimiento INCREMENT pone como máximo un bit, en la línea 6, y por tanto el coste amortizado de una operación INCREMENT es como máximo 2 dólares. El número de 1 en el contador nunca se vuelve negativo y, por lo tanto, la cantidad de crédito permanece no negativa en todo momento. Así, para  $n$  operaciones de INCREMENTO , el costo total amortizado es On/, que acota el costo real total.

### Ejercicios

#### 17.2-1

Suponga que realizamos una secuencia de operaciones de pila en una pila cuyo tamaño nunca excede  $k$ . Después de cada  $k$  operaciones, hacemos una copia de toda la pila para fines de respaldo. Muestre que el costo de  $n$  operaciones de pila, incluida la copia de la pila, es On/ asignando costos amortizados adecuados a las diversas operaciones de pila.

**17.2-2**

Vuelva a hacer el ejercicio 17.1-3 usando un método contable de análisis.

**17.2-3**

Supongamos que deseamos no sólo incrementar un contador sino también restablecerlo a cero (es decir, hacer que todos sus bits sean 0). Contando el tiempo para examinar o modificar un bit como  $.1/$ , muestre cómo implementar un contador como una matriz de bits para que cualquier secuencia de  $n$  operaciones INCREMENT y RESET tome tiempo  $O(n)$  en un contador inicialmente cero. (Sugerencia: mantenga un puntero en el 1 de orden superior).

## 17.3 El método potencial

En lugar de representar el trabajo prepago como crédito almacenado con objetos específicos en la estructura de datos, el método potencial de análisis amortizado representa el trabajo prepago como "energía potencial" o simplemente "potencial", que puede liberarse para pagar operaciones futuras. Asociamos el potencial con la estructura de datos como un todo en lugar de con objetos específicos dentro de la estructura de datos.

El método potencial funciona de la siguiente manera. Realizaremos  $n$  operaciones, comenzando con una estructura de datos inicial  $D_0$ . Para cada  $i \in \{1, 2, \dots, n\}$ , dejamos que  $c_i$  sea el costo real de la  $i$ -ésima operación y que  $D_i$  sea la estructura de datos que resulta después de aplicar la  $i$ -ésima operación a la estructura de datos  $D_{i-1}$ . Una función potencial  $\hat{\cdot}$  asigna cada estructura de datos  $D_i$  a un número real  $\hat{\cdot}.D_i$ , que es el potencial asociado con la estructura de datos  $D_i$ . El costo amortizado  $c_{i+1}$  de la  $i$ -ésima operación con respecto a la función potencial  $\hat{\cdot}$  está definido por

$$c_{i+1} = C \hat{\cdot} D_i / \hat{\cdot} D_{i+1} : \quad (17.2)$$

El coste amortizado de cada operación es, por tanto, su coste real más el cambio de potencial debido a la operación. Por la ecuación (17.2), el costo total amortizado de las  $n$  operaciones es

$$\sum_{i=1}^n c_{i+1} = \sum_{i=1}^n C \hat{\cdot} D_i / \hat{\cdot} D_{i+1}$$

$$\sum_{i=1}^n c_{i+1} = C \hat{\cdot} D_1 / \hat{\cdot} D_n : \quad (17.3)$$

La segunda igualdad se deriva de la ecuación (A.9) porque los términos  $\hat{\cdot} D_i$  son telescopicos.

Si podemos definir una función potencial  $\hat{\cdot}$  tal que  $\hat{\cdot} D_n / \hat{\cdot} D_0$ , entonces el total  $c_{i+1}$  da un costo amortizado  $P_n$  límite superior al costo total real  $P_n$ .

En la práctica, no siempre sabemos cuántas operaciones se pueden realizar. Por lo tanto, si requerimos que  $\hat{D}_i / \hat{D}_0$  para todo  $i$ , entonces garantizamos, como en el método contable, que pagamos por adelantado. Por lo general, simplemente definimos  $\hat{D}_0$  como 0 y luego mostramos que  $\hat{D}_i / 0$  para todo  $i$ . (Consulte el ejercicio 17.3-1 para conocer una forma fácil de manejar casos en los que  $\hat{D}_0 \neq 0$ .)

Intuitivamente, si la diferencia de potencial  $\hat{D}_i / \hat{D}_{i-1}$  de la  $i$ -ésima operación es positiva, entonces el costo amortizado  $c_i$  representa un sobrecargo a la  $i$ -ésima operación, y el potencial de la estructura de datos aumenta. Si la diferencia de potencial es negativa, entonces el costo amortizado representa un cargo inferior a la  $i$ -ésima operación, y la disminución en el potencial paga el costo real de la operación.

Los costos amortizados definidos por las ecuaciones (17.2) y (17.3) dependen de la elección de la función potencial  $\hat{\cdot}$ . Diferentes funciones potenciales pueden generar diferentes costos amortizados y aun así ser límites superiores de los costos reales. A menudo encontramos compensaciones que podemos hacer al elegir una función potencial; la mejor función potencial para usar depende de los límites de tiempo deseados.

#### operaciones de pila

Para ilustrar el método potencial, volvemos una vez más al ejemplo de las operaciones de pila PUSH, POP y MULTIPOP. Definimos la función potencial  $\hat{\cdot}$  en una pila como el número de objetos en la pila. Para la pila vacía  $D_0$  con la que comenzamos, tenemos  $\hat{D}_0 / D_0$ . Dado que el número de objetos en la pila nunca es negativo, la pila  $D_i$  que resulta después de la  $i$ -ésima operación tiene un potencial no negativo y, por lo tanto,

$$\begin{aligned} \hat{D}_i / &= 0 \\ D \hat{D}_0 / : & \end{aligned}$$

El costo total amortizado de  $n$  operaciones con respecto a  $\hat{\cdot}$  representa por lo tanto un límite superior del costo real.

Calculemos ahora los costos amortizados de las diversas operaciones de apilamiento. Si la  $i$ -ésima operación en una pila que contiene  $s$  objetos es una operación EMPUJAR, entonces la diferencia de potencial es

$$\begin{aligned} \hat{D}_i / \hat{D}_{i-1} / D . s C 1 / s \\ D 1 \end{aligned}$$

Por la ecuación (17.2), el costo amortizado de esta operación PUSH es

$$c_i D ci C \hat{D}_i / \hat{D}_{i-1}$$

$$D 1 C 1$$

$$D 2$$

Suponga que la i-ésima operación en la pila es MULTIPOP.S; k/, lo que provoca k0 D min.k; s/ objetos que se sacarán de la pila. El costo real de la operación es k0 , y la diferencia de potencial es

$$\hat{D}_i / \hat{D}_{i-1} / D_{k0}$$

Así, el costo amortizado de la operación MULTIPOP es

$$cyi D ci C \hat{D}_i / \hat{D}_{i-1} / D_{k0} k0$$

$$D_0$$

De manera similar, el costo amortizado de una operación POP ordinaria es 0.

El costo amortizado de cada una de las tres operaciones es O.1/, por lo que el costo total amortizado de una secuencia de n operaciones es On/. Como ya hemos argumentado que  $\hat{D}_i / \hat{D}_{i-1} / D_0$ , el costo total amortizado de n operaciones es un límite superior del costo total real. Por lo tanto, el costo en el peor de los casos de n operaciones es On/.

### Incrementando un contador binario

Como otro ejemplo del método potencial, nuevamente observamos el incremento de un contador binario. Esta vez, definimos que el potencial del contador después de la i-ésima operación de INCREMENTO es bi, el número de 1 en el contador después de la i-ésima operación.

Calculemos el costo amortizado de una operación de INCREMENTO . Suponga que la i-ésima operación de INCREMENTO restablece los bits ti . El costo real de la operación es, por lo tanto, como máximo ti C 1, ya que además de restablecer ti bits, establece como máximo un bit a 1. Si bi D 0, entonces la i-ésima operación restablece todos los k bits, y así bi1 D ti D k. Si bi > 0, entonces bi D bi1 ti C 1. En cualquier caso, bi bi1 ti C 1, y la diferencia de potencial es

$$\begin{aligned} \hat{D}_i / \hat{D}_{i-1} / & \quad .bi1 ti C 1 / bi1 \\ D_1 & \quad ti : \end{aligned}$$

Por lo tanto, el costo amortizado es

$$cyi D ci C \hat{D}_i / \hat{D}_{i-1} / .ti C 1 / C .1$$

$$ti / D 2$$

Si el contador comienza en cero, entonces  $\hat{D}_0 / D_0$ . Dado que  $\hat{D}_i / 0$  para todo i, el costo total amortizado de una secuencia de n operaciones INCREMENT es un límite superior del costo real total, por lo que el peor -el costo por caja de n operaciones INCREMENT es On/.

El método potencial nos brinda una manera fácil de analizar el contador incluso cuando no comienza en cero. El contador comienza con b0 1s, y después de n INCREMENTO

operaciones tiene  $b_n$  1s, donde 0  $b_0$ ;  $b_{n-k}$  (Recuerde que  $k$  es el número de bits en el contador). Podemos reescribir la ecuación (17.3) como

$$\sum_{i=1}^n c_i D \sum_{i=1}^n c_i \cdot D_n / C \cdot D_0 : \quad (17.4)$$

Tenemos  $c_i = 2$  para todo  $i < n$ . Dado que  $\cdot D_0 / D b_0$  y  $\cdot D_n / D b_n$ , el costo real total de  $n$  operaciones INCREMENT es

$$\sum_{i=1}^n c_i \quad \sum_{i=1}^n 2 \text{ mil millones } C b_0$$

$$D 2n b_n C b_0 :$$

Obsérvese en particular que, dado que  $b_0 = k$ , mientras  $k \leq O(n)$ , el costo real total es  $O(n)$ . En otras palabras, si ejecutamos al menos  $n$   $D/k$  operaciones de INCREMENTO, el costo real total es  $O(n)$ , sin importar el valor inicial que contenga el contador.

### Ejercicios

#### 17.3-1

Suponga que tenemos una función potencial  $\hat{}$  tal que  $\hat{D}_i / \hat{D}_0$  para todo  $i$ , pero  $\hat{D}_0 = 0$ . Demuestre que existe una función potencial  $\hat{0}$  tal que  $\hat{0} \cdot D_0 / D 0$ ,  $\hat{0} \cdot D_i / 0$  para todo  $i \geq 1$ , y los costos amortizados usando  $\hat{0}$  son los mismos que los costos amortizados usando  $\hat{}$ .

#### 17.3-2

Vuelva a hacer el ejercicio 17.1-3 utilizando un posible método de análisis.

#### 17.3-3

Considere una estructura binaria ordinaria de datos min-heap con  $n$  elementos que soportan las instrucciones INSERT y EXTRACT-MIN en  $O(\lg n)$  en el peor de los casos. Dé una función potencial  $\hat{}$  tal que el costo amortizado de INSERT sea  $O(\lg n)$  y el costo amortizado de EXTRACT-MIN sea  $O(1)$ , y demuestre que funciona.

#### 17.3-4

¿Cuál es el costo total de ejecutar  $n$  de las operaciones de la pila PUSH, POP y MULTIPOP, suponiendo que la pila comienza con  $s_0$  objetos y termina con  $s_n$  objetos?

#### 17.3-5

Suponga que un contador comienza en un número con  $b$  1s en su representación binaria, en lugar de 0. Demuestre que el costo de realizar  $n$  operaciones de INCREMENTO es  $O(n)$  si  $b = O(1)$ . (No suponga que  $b$  es constante).

## 17.3-6

Muestre cómo implementar una cola con dos pilas ordinarias (Ejercicio 10.1-6) de modo que el costo amortizado de cada operación ENQUEUE y cada DEQUEUE sea O.1/.

## 17.3-7

Diseñe una estructura de datos que admita las siguientes dos operaciones para un multiconjunto dinámico S de enteros, que permita valores duplicados:

INSERTAR.S; x/ inserta x en S.

DELETE-LARGER-HALF.S / elimina los elementos  $djSj = 2e$  más grandes de S.

Explique cómo implementar esta estructura de datos para que cualquier secuencia de operaciones m INSERT y DELETE-LARGER-HALF se ejecute en  $O(m \cdot \log n)$  tiempo. Su implementación también debe incluir una forma de generar los elementos de S en tiempo  $O.jSj/$ .

## 17.4 Tablas dinámicas

No siempre sabemos de antemano cuántos objetos almacenarán algunas aplicaciones en una tabla. Podríamos asignar espacio para una mesa, solo para descubrir más tarde que no es suficiente. Luego debemos reasignar la tabla con un tamaño mayor y copiar todos los objetos almacenados en la tabla original en la nueva tabla más grande. De manera similar, si se han eliminado muchos objetos de la tabla, puede valer la pena reasignar la tabla con un tamaño más pequeño. En esta sección, estudiaremos este problema de expandir y contraer dinámicamente una tabla. Usando el análisis amortizado, mostraremos que el costo amortizado de inserción y eliminación es solo O.1/, aunque el costo real de una operación es grande cuando desencadena una expansión o una contracción. Además, veremos cómo garantizar que el espacio no utilizado en una tabla dinámica nunca supere una fracción constante del espacio total.

Suponemos que la tabla dinámica admite las operaciones TABLE-INSERT y TABLE-DELETE. TABLE-INSERT inserta en la tabla un elemento que ocupa un solo espacio, es decir, un espacio para un elemento. Del mismo modo, TABLE-DELETE elimina un elemento de la tabla, liberando así un espacio. Los detalles del método de estructuración de datos utilizado para organizar la tabla no son importantes; podríamos usar una pila (Sección 10.1), un montón (Capítulo 6) o una tabla hash (Capítulo 11). También podríamos usar un arreglo o colección de arreglos para implementar el almacenamiento de objetos, como hicimos en la Sección 10.3.

Encontraremos conveniente utilizar un concepto introducido en nuestro análisis de hashing (Capítulo 11). Definimos el factor de carga  $\alpha$  de una tabla no vacía T como el número de elementos almacenados en la tabla dividido por el tamaño (número de ranuras) de la tabla. Asignamos una tabla vacía (una sin elementos) de tamaño 0 y definimos su factor de carga como 1. Si el factor de carga de una tabla dinámica está limitado por una constante,

el espacio no utilizado en la tabla nunca es más que una fracción constante de la cantidad total de espacio.

Comenzamos analizando una tabla dinámica en la que solo insertamos elementos. Nosotros entonces considere el caso más general en el que insertamos y eliminamos elementos.

#### 17.4.1 Ampliación de tabla

Supongamos que el almacenamiento de una tabla se asigna como una matriz de ranuras. Una mesa se llena cuando se han utilizado todas las ranuras o, de manera equivalente, cuando su factor de carga es  $\frac{1}{2}$ . En algunos entornos de software, al intentar insertar un elemento en una tabla completa, la única alternativa es abortar con un error. Supondremos, sin embargo, que nuestro entorno de software, como muchos de los modernos, proporciona un sistema de gestión de memoria que puede asignar y liberar bloques de almacenamiento a petición. Por lo tanto, al insertar un elemento en una tabla completa, podemos expandir la tabla asignando una nueva tabla con más ranuras que las que tenía la tabla anterior. Debido a que siempre necesitamos que la tabla resida en la memoria contigua, debemos asignar una nueva matriz para la tabla más grande y luego copiar elementos de la tabla anterior a la nueva tabla.

Una heurística común asigna una nueva tabla con el doble de ranuras que la anterior. Si las únicas operaciones de la tabla son las inserciones, entonces el factor de carga de la tabla siempre es al menos  $1=2$  y, por lo tanto, la cantidad de espacio desperdiciado nunca supera la mitad del espacio total de la tabla.

En el siguiente pseudocódigo, asumimos que T es un objeto que representa la tabla. El atributo T:table contiene un puntero al bloque de almacenamiento que representa la tabla, T:num contiene el número de elementos de la tabla y T:size proporciona el número total de espacios en la tabla. Inicialmente, la tabla está vacía: T:num D T:size D 0.

```
MESA-INSERCIÓN.T; x/ 1 si
    T:tamaño == 0 asignar
    1           T:tabla con 1 ranura 2 3 T:tamaño D
    4 si T:num == T:tamaño 5
        asignar nueva tabla con 2
        ranuras T:tamaño 6 insertar todos los elementos en T:table
        into new-table 7 free T:table 8 T:table D new-table 9 T:size D 2
        T:size 10 insert x into
        T:table 11 T:num D T:num C 1
```

---

<sup>1</sup> En algunas situaciones, como una tabla hash de dirección abierta, es posible que deseemos considerar que una tabla está llena si su factor de carga es igual a alguna constante estrictamente menor que 1. (Consulte el ejercicio 17.4-1).

Observe que aquí tenemos dos procedimientos de "inserción": el procedimiento TABLE-INSERT en sí mismo y la inserción elemental en una tabla en las líneas 6 y 10. Podemos analizar el tiempo de ejecución de TABLE-INSERT en términos del número de inserciones elementales por asignando un costo de 1 a cada inserción elemental. Suponemos que el tiempo de ejecución real de TABLE-INSERT es lineal en el tiempo para insertar elementos individuales, de modo que la sobrecarga para asignar una tabla inicial en la línea 2 es constante y domina la sobrecarga para asignar y liberar almacenamiento en las líneas 5 y 7 por el costo de transferir artículos en la línea 6. Llamamos expansión al evento en el que se ejecutan las líneas 5 a 9 .

Analicemos una secuencia de  $n$  operaciones TABLE-INSERT en una tabla inicialmente vacía. ¿ Cuál es el costo  $c_i$  de la  $i$ -ésima operación? Si la tabla actual tiene espacio para el nuevo elemento (o si esta es la primera operación), entonces  $c_i = D_1$ , ya que solo necesitamos realizar una inserción elemental en la línea 10. Sin embargo, si la tabla actual está llena y una expansión ocurre, entonces  $c_i = D_i$ : el costo es 1 para la inserción elemental en la línea 10 más 1 para los elementos que debemos copiar de la tabla antigua a la nueva tabla en la línea 6. Si realizamos  $n$  operaciones, el peor de los casos el costo de una operación es  $O(n)$ , lo que conduce a un límite superior de  $O(n^2)$  en el tiempo total de ejecución de  $n$  operaciones.

Este límite no es estricto, porque rara vez expandimos la tabla en el curso de  $n$  operaciones TABLE-INSERT . Específicamente, la  $i$ -ésima operación causa una expansión solo cuando  $i$  1 es una potencia exacta de 2. El costo amortizado de una operación es de hecho  $O(1)$ , como podemos demostrar utilizando el análisis agregado. El costo de la  $i$ -ésima operación es

1 es una potencia exacta de 2 :

$c_i = D_1$  (yo si yo<sup>1</sup> de lo contrario:

Por lo tanto , el costo total de  $n$  operaciones TABLE-INSERT es

$$\begin{array}{ll} X_n & c_i \\ \text{norte} & X \\ iD_1 & jD_0 \end{array}$$

b lg nc

$$2^j$$

$$< n C 2n$$

$$D 3n$$

:

porque como mucho  $n$  operaciones cuestan 1 y los costos de las operaciones restantes forman una serie geométrica. Dado que el costo total de  $n$  operaciones TABLE-INSERT está limitado por  $3n$ , el costo amortizado de una sola operación es como máximo 3.

Al usar el método de contabilidad, podemos tener una idea de por qué el costo amortizado de una operación TABLE-INSERT debe ser 3. Intuitivamente, cada elemento paga por 3 inserciones elementales: insertarse en la tabla actual, moverse cuando la tabla se expande y mover otro elemento que ya se ha movido una vez cuando se expande la tabla. Por ejemplo, suponga que el tamaño de la tabla es  $m$  inmediatamente después de una expansión.

Entonces la tabla contiene  $m = 2$  elementos y contiene

sin crédito. Cobramos 3 dólares por cada inserción. La inserción elemental que se produce inmediatamente cuesta 1 dólar. Colocamos otro dólar como crédito en el artículo insertado. Colocamos el tercer dólar como crédito en uno de los  $m=2$  artículos que ya están en la tabla. La mesa no se volverá a llenar hasta que hayamos insertado otros  $m=2$  1 elementos y, por lo tanto, cuando la mesa contenga  $m$  elementos y esté llena, habremos colocado un dólar en cada elemento para pagar por reinsertarlo durante la expansión.

Podemos usar el método potencial para analizar una secuencia de  $n$  operaciones TABLE-INSERT , y lo usaremos en la Sección 17.4.2 para diseñar una operación TABLE-DELETE que también tiene un costo amortizado de  $O(1)$ . Comenzamos definiendo una función de potencial  $\hat{\cdot}$  que es 0 inmediatamente después de una expansión, pero que alcanza el tamaño de la tabla cuando la mesa está llena, de modo que podemos pagar la próxima expansión según el potencial. La función

$\hat{\cdot}T / D 2 T:\text{num} T:\text{size} (17.5)$  es una posibilidad. Inmediatamente después de una expansión, tenemos  $T:\text{num} D T:\text{size}=2$ , y por lo tanto  $\hat{\cdot}T / D 0$ , como se desea. Inmediatamente antes de una expansión, tenemos  $T:\text{num} D T:\text{size}$ , y por lo tanto  $\hat{\cdot}T / D T:\text{num}$ , como se deseé. El valor inicial del potencial es 0, y dado que la tabla siempre está llena por lo menos hasta la mitad,  $T:\text{num} T:\text{size}=2$ , lo que implica que  $\hat{\cdot}T /$  siempre es no negativo. Por lo tanto, la suma de los costos amortizados de  $n$  operaciones TABLE-INSERT da un límite superior a la suma de los costos reales.

Para analizar el costo amortizado de la  $i$ -ésima operación TABLE-INSERT , dejamos que  $\text{num}_i$  denote el número de elementos almacenados en la tabla después de la  $i$ -ésima operación,  $\text{size}_i$  denote el tamaño total de la tabla después de la  $i$ -ésima operación, y  $\hat{\cdot}^i$  denote el potencial después de la  $i$ -ésima operación.  $i$ -ésima operación. Inicialmente, tenemos  $\text{num}_0 D 0$ ,  $\text{size}_0 D 0$  y  $\hat{\cdot}^0 D 0$ .

Si la  $i$ -ésima operación TABLE-INSERT no desencadena una expansión, entonces tenemos  $\text{tamaño}_i D \text{tamaño}_{i-1}$  y el costo amortizado de la operación es  $\text{cyi} D \text{ci} C \hat{\cdot}^i$

$$\hat{\cdot}^{i-1} D 1 C .2 \text{ num}_i \text{tamaño}_i / .2 \text{ num}_i \text{tamaño}_i / D 1 C .$$

$$2 \text{ num}_i \text{tamaño}_i / .2 \text{ num}_i 1 / \text{tamaño}_i /$$

$$D 3$$

Si la  $i$ -ésima operación desencadena una expansión, entonces tenemos  $\text{size}_i D 2 \text{ size}_{i-1}$  y  $\text{size}_i D \text{num}_i D \text{num}_{i-1}$ , lo que implica que  $\text{size}_i D 2 \text{ num}_i 1 /$ . Así, el costo amortizado de la operación es  $\text{cyi} D \text{ci} C \hat{\cdot}^i \hat{\cdot}^{i-1} D \text{ num}_i C .2 \text{ num}_i \text{tamaño}_i /$

$$.2 \text{ num}_i \text{tamaño}_i / D \text{ num}_i C .2 \text{ num}_i 2 \text{ num}_i 1 / .2 \text{ num}_i 1 / .2 \text{ num}_i 1 / D \text{ num}_i C 2 \text{ num}_i 1 /$$

$$D 3$$

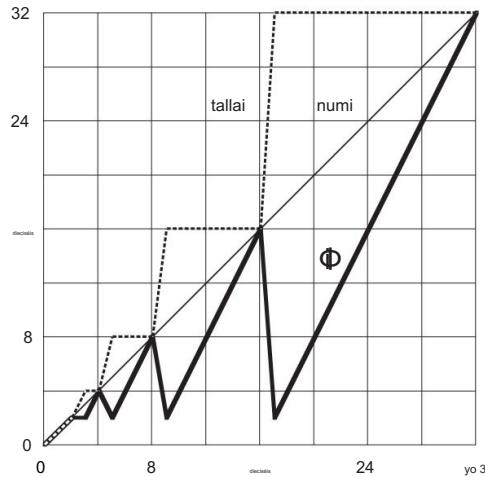


Figura 17.3 El efecto de una secuencia de  $n$  operaciones TABLE-INSERT sobre el número numi de elementos en la tabla, el número sizei de ranuras en la tabla y el potencial  $\Phi$  de  $D_2$  numi sizei , cada uno medido después de la  $i$ -ésima operación. La línea delgada muestra numi , la línea discontinua muestra sizei y la línea gruesa muestra  $\Phi$ . Tenga en cuenta que inmediatamente antes de una expansión, el potencial se ha acumulado hasta la cantidad de elementos en la tabla y, por lo tanto, puede pagar por mover todos los elementos a la nueva tabla. Posteriormente, el potencial cae a 0, pero inmediatamente se incrementa en 2 al insertar el elemento que provocó la expansión.

La figura 17.3 traza los valores de numi , sizei y  $\Phi$  contra  $i$ . Observe cómo se acumula el potencial para pagar por expandir la tabla.

#### 17.4.2 Expansión y contracción de la mesa

Para implementar una operación TABLE-DELETE , es bastante simple eliminar el elemento especificado de la tabla. Sin embargo, para limitar la cantidad de espacio desperdiciado, es posible que deseemos contraer la mesa cuando el factor de carga sea demasiado pequeño. La contracción de la tabla es análoga a la expansión de la tabla: cuando el número de elementos de la tabla es demasiado bajo, asignamos una nueva tabla más pequeña y luego copiamos los elementos de la tabla anterior a la nueva. Entonces podemos liberar el almacenamiento de la tabla antigua devolviéndola al sistema de administración de memoria. Idealmente, nos gustaría preservar dos propiedades:

el factor de carga de la tabla dinámica está acotado por debajo por una constante positiva, y

el costo amortizado de una operación de mesa está acotado arriba por una constante.

Suponemos que medimos el costo en términos de inserciones y eliminaciones elementales.

Podría pensar que deberíamos duplicar el tamaño de la tabla al insertar un elemento en una tabla completa y reducir a la mitad el tamaño cuando eliminar un elemento haría que la mesa se llenara menos de la mitad. Esta estrategia garantizaría que el factor de carga de la tabla nunca caiga por debajo de  $1=2$ , pero lamentablemente puede hacer que el costo amortizado de una operación sea bastante grande. Considere el siguiente escenario. Realizamos  $n$  donde  $n$  es una potencia exacta de 2. Las primeras  $n=2$  nuestro análisis anterior , operaciones son operaciones en una tabla T inserciones, que según cuestan un total de  $.n/$ . Al final de esta secuencia de inserciones,  $T:\text{num } D \ T:\text{size } D \ n=2$ . Para las segundas  $n=2$  operaciones, realizamos la siguiente secuencia:

insertar, eliminar, eliminar, insertar, insertar, eliminar, eliminar, insertar, insertar, insertar, . . .

La primera inserción hace que la tabla se expanda al tamaño  $n$ . Las dos supresiones siguientes hacen que la tabla se contraiga de nuevo al tamaño  $n=2$ . Dos inserciones más provocan otra expansión, y así sucesivamente. El costo de cada expansión y contracción es  $.n/$ , y hay  $.n/$  de ellas. Así, el coste total de las  $n$  operaciones es  $.n2/$ , siendo el coste amortizado de una operación  $.n/$ .

La desventaja de esta estrategia es obvia: después de expandir la tabla, no eliminamos suficientes elementos para pagar una contracción. Asimismo, después de contraer la mesa, no insertamos elementos suficientes para pagar una expansión.

Podemos mejorar esta estrategia al permitir que el factor de carga de la tabla caiga por debajo de  $1=2$ . Específicamente, continuamos duplicando el tamaño de la tabla al insertar un elemento en una tabla completa, pero reducimos a la mitad el tamaño de la tabla cuando se elimina un elemento, lo que hace que la tabla se llene menos de  $1 = 4$ , en lugar de  $1 = 2$  como antes. Por lo tanto, el factor de carga de la tabla está limitado por la constante  $1=4$ .

Intuitivamente, consideraríamos un factor de carga de  $1=2$  como ideal, y el potencial de la mesa sería entonces 0. A medida que el factor de carga se desvía de  $1=2$ , el potencial aumenta de modo que en el momento en que expandimos o contraemos la mesa, la mesa ha acumulado suficiente potencial para pagar por copiar todos los elementos en la mesa recién asignada.

Por lo tanto, necesitaremos una función potencial que haya crecido a  $T:\text{num}$  en el momento en que el factor de carga haya aumentado a 1 o disminuido a  $1=4$ . Después de expandir o contraer la mesa, el factor de carga vuelve a  $1=2$  y el potencial de la mesa se reduce a 0.

Omitimos el código para TABLE-DELETE, ya que es análogo a TABLE-INSERT.

Para nuestro análisis, supondremos que cada vez que el número de elementos en la tabla cae a 0, liberamos el almacenamiento para la tabla. Es decir, si  $T:\text{num } D \ 0$ , entonces  $T:\text{size } D \ 0$ .

Ahora podemos usar el método potencial para analizar el costo de una secuencia de  $n$  operaciones TABLE-INSERT y TABLE-DELETE . Comenzamos definiendo una función de potencial  $\hat{\cdot}$  que es 0 inmediatamente después de una expansión o contracción y aumenta a medida que el factor de carga aumenta a 1 o disminuye a  $1=4$ . Denotemos el factor de carga

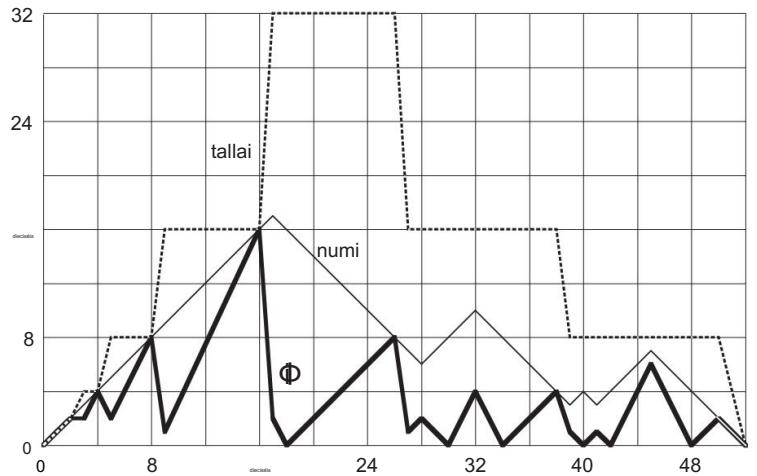


Figura 17.4 El efecto de una secuencia de  $n$  operaciones TABLE-INSERT y TABLE-DELETE en el número  $\text{num}_i$  de elementos en la tabla, el número  $\text{size}_i$  de ranuras en la tabla y el potencial

$\hat{i} D = \begin{cases} 2 \text{ num}_i \text{ size}_i & \text{si } i=1=2 ; \text{ tamaño}_i \\ =2 \text{ num}_i & \text{si } i < 1=2 ; \end{cases}$

cada uno medido después de la  $i$ -ésima operación. La línea delgada muestra  $\text{num}_i$ , la línea discontinua muestra  $\text{size}_i$  y la línea gruesa muestra  $\hat{i}$ . Tenga en cuenta que inmediatamente antes de una expansión, el potencial se ha acumulado hasta la cantidad de elementos en la tabla y, por lo tanto, puede pagar por mover todos los elementos a la nueva tabla. Asimismo, inmediatamente antes de una contracción, el potencial se ha acumulado hasta el número de elementos de la tabla.

tor de una tabla no vacía  $T$  por  $\text{.T} / D T:\text{num}=T:\text{size}$ . Dado que para una tabla vacía,  $T:\text{num} D T:\text{size} D 0$  y  $\text{.T} / D 1$ , siempre tenemos  $T:\text{num} D \text{.T} / T:\text{size}$ , ya sea que la tabla esté vacía o no. Usaremos como nuestra función potencial

$$\hat{\text{.T}} / D (2 T:\text{tamaño}-2 T:\text{num}) \text{ si } 1=2 : \quad (17.6)$$

Observe que el potencial de una mesa vacía es 0 y que el potencial nunca es negativo. Por lo tanto, el costo total amortizado de una secuencia de operaciones con respecto a  $\hat{i}$  proporciona un límite superior para el costo real de la secuencia.

Antes de proceder con un análisis preciso, hacemos una pausa para observar algunas propiedades de la función de potencial, como se ilustra en la figura 17.4. Observe que cuando el factor de carga es  $1=2$ , el potencial es 0. Cuando el factor de carga es 1, tenemos  $T:\text{size} D T:\text{num}$ , lo que implica  $\text{.T} / D T:\text{num}$ , y por lo tanto el potencial puede pagar para una expansión si se inserta un artículo. Cuando el factor de carga es  $1=4$ , tenemos  $T:\text{size} D 4T:\text{num}$ , que

implica  $\hat{C} \cdot T / D \leq T : \text{num}$  y, por lo tanto, el potencial puede compensar una contracción si se elimina un elemento.

Para analizar una secuencia de  $n$  operaciones INSERTAR-TABLA y ELIMINAR-TABLA, hacemos que  $c_i$  denote el costo real de la  $i$ -ésima operación,  $\tilde{c}_i$  denote su costo amortizado con respecto a  $\hat{C}$ ,  $\text{num}_i$  denote el número de elementos almacenados en la tabla después de la  $i$ -ésima operación,  $\text{size}_i$  denota el tamaño total de la mesa después de la  $i$ -ésima operación,  $\lambda_i$  denota el factor de carga de la mesa después de la  $i$ -ésima operación, y  $\tilde{\lambda}_i$  denota el potencial después de la  $i$ -ésima operación. Inicialmente,  $\text{num}_0 = 0$ ,  $\text{size}_0 = 0$ ,  $\lambda_0 = 1$  y  $\tilde{\lambda}_0 = 0$ .

Empezamos con el caso en el que la  $i$ -ésima operación es TABLE-INSERT. El análisis es idéntico al de la expansión de la tabla en la Sección 17.4.1 si  $\lambda_i < 1=2$ . Ya sea que la tabla se expanda o no, el costo amortizado  $\tilde{c}_i$  de la operación es como máximo 3.

Si  $\lambda_i < 1=2$ , la tabla no se puede expandir como resultado de la operación, ya que la tabla se expande solo cuando  $\lambda_i \geq 1=2$ . Si  $\lambda_i < 1=2$  también, entonces el costo amortizado de la  $i$ -ésima operación es  $\tilde{c}_i = D$ .

$$\tilde{c}_i \geq 1$$

$$\begin{aligned} D &\leq C \cdot \text{size}_i = 2 \cdot \text{num}_i / \text{size}_i = 2 \cdot \text{num}_i / (D + C \cdot \text{size}_i) \\ &\leq 2 \cdot \text{num}_i / (D + 2 \cdot \text{num}_i) \leq D \end{aligned}$$

Si  $\lambda_i < 1=2$  pero  $\lambda_{i+1} = 2$ , entonces  $\tilde{c}_i$

$$D \leq c_i + \tilde{c}_{i+1}$$

$$D \leq C \cdot 2 \cdot \text{num}_i / \text{size}_i = 2 \cdot \text{num}_i / (D + C \cdot \text{size}_i)$$

$$D \leq 2 \cdot \text{num}_i / (\text{size}_i + 2 \cdot \text{num}_i) \leq D$$

$$D \leq 3 \cdot \frac{\text{tamaño}_i}{2} \leq 3 \cdot \text{tamaño}_i$$

$$D \leq 3 \cdot \frac{\text{tamaño}_{i+1}}{2} \leq 3 \cdot \text{tamaño}_{i+1}$$

$$D \leq \frac{3}{2} \cdot \text{tamaño}_i \leq \frac{3}{2} \cdot \text{tamaño}_{i+1}$$

$$D \leq 3$$

Así, el coste amortizado de una operación TABLE-INSERT es como máximo 3.

Pasamos ahora al caso en el que la  $i$ -ésima operación es TABLE-DELETE. En este caso,  $\text{num}_i = \text{num}_{i+1} - 1$ . Si  $\lambda_i < 1=2$ , entonces debemos considerar si la operación hace que la mesa se contraiga. Si no es así, entonces  $\text{size}_i = \text{size}_{i+1} + 1$  y el costo amortizado de la operación es  $\tilde{c}_i = D$ .

$$\tilde{c}_i = D \leq C \cdot \text{size}_i = 2 \cdot \text{num}_i / \text{size}_i$$

$$D \leq 2 \cdot \text{num}_i / (\text{size}_i + 1) \leq D$$

Si  $j_1 < i=2$  y la  $i$ -ésima operación desencadena una contracción, entonces el costo real de la operación es  $c_i D \text{ numi} C 1$ , ya que eliminamos un elemento y movemos  $\text{numi}$  elementos.

Tenemos  $\text{sizei}=2 D \text{ sizei1}=4 D \text{ numi1} D \text{ numi} C 1$ , y el costo amortizado de la operación es

$c_{yi} D c_i C \hat{i} i_1 D .\text{numi} C 1 /$

$C .\text{sizei}=2 \text{ numi} / .\text{sizei1}=2 \text{ numi1} / D .\text{numi} C 1 / C ..\text{numi} C 1 / \text{numi} / ..2$

$\text{numi} C 2 / .\text{numi} C 1 //$

$D 1$

Cuando la  $i$ -ésima operación es TABLE-DELETE y  $j_1 = 2$ , el costo amortizado también está acotado arriba por una constante. Dejamos el análisis como Ejercicio 17.4-2.

En resumen, dado que el costo amortizado de cada operación está acotado arriba por una constante, el tiempo real para cualquier secuencia de  $n$  operaciones en una tabla dinámica es  $O(n)$ .

## Ejercicios

### 17.4-1

Suponga que deseamos implementar una tabla hash dinámica de direcciones abiertas. ¿Por qué podríamos considerar que la tabla está llena cuando su factor de carga alcanza algún valor  $\alpha$  que es estrictamente menor que 1? Describa brevemente cómo hacer que la inserción en una tabla hash dinámica de direcciones abiertas se ejecute de tal manera que el valor esperado del costo amortizado por inserción sea  $O(1)$ . ¿Por qué el valor esperado del costo real por inserción no es necesariamente  $O(1)$  para todas las inserciones?

### 17.4-2

que si  $j_1$  ELIMINAR,  $i=2$  y la  $i$ -ésima operación en una tabla dinámica es TABLA Demuestre entonces el costo amortizado de la operación con respecto a la función potencial (17.6) está acotado arriba por una constante.

### 17.4-3

Suponga que en lugar de contraer una mesa reduciendo su tamaño a la mitad cuando su factor de carga cae por debajo de  $1/4$ , la contraemos multiplicando su tamaño por  $2/3$  cuando su factor de carga cae por debajo de  $1/3$ . Usando la función potencial

$^T / D j_2 T:\text{numi} T:\text{tamaño} ;$

muestre que el costo amortizado de un TABLE-DELETE que usa esta estrategia está acotado arriba por una constante.

## Problemas

### 17-1 Contador binario de bits invertidos El

capítulo 30 examina un algoritmo importante llamado transformada rápida de Fourier o FFT. El primer paso del algoritmo FFT realiza una permutación de inversión de bits en  $n_1$  cuya longitud es  $A \leq n_1 < 2^k$ . Esta permutación intercambia elementos cuyos índices tienen representaciones binarias que son inversas entre sí.

Podemos expresar cada índice  $a$  como una secuencia de  $k$  bits  $a_k a_{k-1} \dots a_0$ , donde  $a = a_k 2^{k-1} + a_{k-1} 2^{k-2} + \dots + a_1 2^0 + a_0 2^{-1}$ .

$\text{revk}(a) = a_k a_{k-1} \dots a_0$

de este modo,

$k$

$\text{revk}(a) = a_k a_{k-1} \dots a_0$

$iD_0$

Por ejemplo, si  $n = 16$  ( $k = 4$ ), entonces  $\text{revk}(3) = 12$ , ya que la representación de 4 bits de 3 es 0011, que cuando se invierte da 1100, la representación de 4 bits de 12.

- Dada una función  $\text{revk}$  que se ejecuta en tiempo  $O(k)$ , escriba un algoritmo para realizar la permutación de inversión de bits en un arreglo de longitud  $n = 2^k$  en tiempo  $O(nk)$ .

Podemos usar un algoritmo basado en un análisis amortizado para mejorar el tiempo de ejecución de la permutación de inversión de bits. Mantenemos un "contador de bits invertidos" y un procedimiento BIT-INVERTIDO-INCREMENTO que, cuando se le da un valor de contador de bits invertidos  $a$ , produce  $\text{revk}(\text{revk}(a))$ . Si  $k = 4$ , por ejemplo, y el contador de bits invertidos comienza en 0, entonces las sucesivas llamadas a BIT-REVERSED-INCREMENT producen la secuencia

0000; 1000; 0100; 1100; 0010; 1010; : : : D 0; 8; 4; 12; 2; 10; : : :

- Suponga que las palabras en su computadora almacenan valores de  $k$  bits y que, en la unidad de tiempo, su computadora puede manipular los valores binarios con operaciones tales como desplazamiento hacia la izquierda o hacia la derecha en cantidades arbitrarias, AND bit a bit, OR bit a bit, etc. Describa una implementación del procedimiento BIT-REVERSED-INCREMENT que permite realizar la permutación de inversión de bits en un arreglo de  $n$  elementos en un tiempo total de  $O(n)$ .

- Suponga que puede desplazar una palabra hacia la izquierda o hacia la derecha solo un bit por unidad de tiempo. ¿Todavía es posible implementar una permutación de inversión de bits  $O(n)$ ?

### 17-2 Dinamización de la búsqueda binaria La

búsqueda binaria de una matriz ordenada requiere un tiempo de búsqueda logarítmico, pero el tiempo para insertar un nuevo elemento es lineal en el tamaño de la matriz. Podemos mejorar el tiempo de inserción manteniendo varias matrices ordenadas.

Especificamente, supongamos que deseamos admitir BUSCAR e INSERTAR en un conjunto de  $n$  elementos. Sea  $k \in \{0, 1, \dots, n\}$ , y sea  $h_{nk}$  la representación binaria de  $n$ ;  $n_0; \dots; n_k$ . Tenemos  $k$  arreglos ordenados  $A_0; A_1; \dots; A_k$ , donde para  $k \geq 1$ , la longitud de la matriz  $A_i$  es  $2^i$  y  $D_0 = 1; \dots; D_{2^k-1}$  dependiendo de  ~~cada matriz tiene su propia longitud~~, respectivamente. El número total de elementos

mentos mantenidos en todos los arreglos  $k$  es, por tanto,  $\sum_{i=0}^k 2^i = 2^{k+1} - 1$ . Aunque cada matriz individual está ordenada, los elementos de diferentes matrices no guardan una relación particular entre sí.

a. Describa cómo realizar la operación de BÚSQUEDA para esta estructura de datos. Analizar su tiempo de ejecución en el peor de los casos.

b. Describa cómo realizar la operación INSERTAR . Analice su peor caso y los tiempos de ejecución amortizados.

C. Discuta cómo implementar DELETE.

### 17-3 Árboles compensados por pesos amortizados

Considere un árbol de búsqueda binario ordinario aumentado agregando a cada nodo  $x$  el atributo  $x:\text{size}$  dando el número de claves almacenadas en el subárbol con raíz en  $x$ . Sea  $c$  una constante en el rango  $1 \leq c < 2$ . Decimos que un nodo dado  $x$  está balanceado en  $c$  si  $x:\text{left}:\text{size} / x:\text{size} \leq c$ . El árbol como un todo es ~~está equilibrado~~, si el ~~equilibrio~~ de cada nodo del árbol está equilibrado en  $c$ . G. Varghese sugirió el siguiente enfoque amortizado para mantener árboles con peso equilibrado.

a. Un árbol balanceado  $c=2$  es, en cierto sentido, tan balanceado como puede ser. Dado un nodo  $x$  en un árbol de búsqueda binario arbitrario, muestre cómo reconstruir el subárbol con raíz en  $x$  para que se equilibre en  $c=2$ . Su algoritmo debe ejecutarse en el tiempo  $O(x:\text{size})$ , y puede usar el almacenamiento auxiliar  $O(x:\text{size})$ .

b. Muestre que realizar una búsqueda en un árbol de búsqueda binario balanceado de  $n$  nodos toma  $O(\lg n)$  tiempo en el peor de los casos.

Para el resto de este problema, suponga que la constante  $c$  es estrictamente mayor que  $1=2$ .

Supongamos que implementamos INSERTAR y ELIMINAR como de costumbre para un árbol de búsqueda binaria de  $n$  nodos, excepto que después de cada operación de este tipo, si algún nodo del árbol ya no está equilibrado en  $c$ , entonces "reconstruimos" el subárbol enraizado en el más alto tal nodo en el árbol para que se convierta en  $c=2$ -equilibrado.

Analizaremos este esquema de reconstrucción utilizando el método potencial. Para un nodo  $x$  en un árbol de búsqueda binaria  $T$ , definimos

$.x/ D jx:izquierda:tamaño x:derecha:tamaño j ;$

y definimos el potencial de  $T$  como

$$\begin{array}{ll} \hat{T} / D c X & .X/ ; \\ & x^2 T \text{ anchox}/2 \end{array}$$

donde  $c$  es una constante suficientemente grande que depende de  $\_$ .

C. Argumente que cualquier árbol de búsqueda binario tiene un potencial no negativo y que un 1=2-árbol equilibrado tiene potencial 0.

d. Suponga que  $m$  unidades de potencial pueden pagar por reconstruir un subárbol de  $m$  nodos.

¿Qué tan grande debe ser  $c$  en términos de  $\_$  para que tome  $O(1/\text{tiempo amortizado})$  para reconstruir un subárbol que no está balanceado en  $\_$ ?

mi. Demuestre que insertar un nodo o eliminar un nodo de un árbol balanceado de  $n$  nodos cuesta  $O(\lg n)$  tiempo amortizado.

#### 17-4 El costo de reestructurar árboles rojo-negro

Hay cuatro operaciones básicas en los árboles rojo-negro que realizan modificaciones estructurales: inserciones de nodos, eliminaciones de nodos, rotaciones y cambios de color. Hemos visto que RB-INSERT y RB-DELETE usan solo rotaciones  $O(1)$ , inserciones de nodos y eliminaciones de nodos para mantener las propiedades rojo-negro, pero pueden hacer muchos más cambios de color.

a. Describa un árbol rojo-negro legal con  $n$  nodos de modo que llamar a RB-INSERT para agregar el  $.n$  C 1/st nodo provoque cambios de color en  $\lg n$ . Luego describa un árbol rojo-negro legal con  $n$  nodos para los cuales llamar a RB-DELETE en un nodo en particular provoca cambios de color  $\lg n$ .

Aunque el número de cambios de color por operación en el peor de los casos puede ser logarítmico, probaremos que cualquier secuencia de  $m$  RB-INSERT y RB-DELETE operaciones en un árbol rojo-negro inicialmente vacío provoca  $O(m)$  modificaciones estructurales en el peor de los casos. Tenga en cuenta que contamos cada cambio de color como una modificación estructural.

b. Algunos de los casos manejados por el bucle principal del código tanto de RB-INSERT FIXUP como de RB-DELETE-FIXUP están terminando: una vez encontrados, hacen que el bucle termine después de un número constante de operaciones adicionales. Para cada uno de los casos de RB-INSERT-FIXUP y RB-DELETE-FIXUP, especificar cuáles son terminantes y cuáles no. (Sugerencia: mire las figuras 13.5, 13.6 y 13.7).

Analizaremos primero las modificaciones estructurales cuando sólo se realizan inserciones.

Sea  $T$  un árbol rojo-negro y defina  $\hat{T}$  como el número de nodos rojos en  $T$ . Suponga que 1 unidad de potencial puede pagar las modificaciones estructurales realizadas por cualquiera de los tres casos de RB-INSERT-FIXUP.

C. Sea  $T'$  ser el resultado de aplicar el Caso 1 de RB-INSERT-FIXUP a  $T$ . Argumentar que  $\hat{T}' \leq \hat{T} - 1$ .

d. Cuando insertamos un nodo en un árbol rojo-negro usando RB-INSERT, podemos dividir la operación en tres partes. Enumere las modificaciones estructurales y los posibles cambios que resultan de las líneas 1 a 16 de RB-INSERT, de los casos de no terminación de RB-INSERT-FIXUP y de los casos de terminación de RB-INSERT-FIXUP.

mi. Utilizando el inciso d), argumente que el número amortizado de modificaciones estructurales formado por cualquier llamada de RB-INSERT es  $O(1)$ .

Ahora deseamos probar que hay modificaciones estructurales  $O(m)$  cuando hay tanto inserciones como eliminaciones. Definamos, para cada nodo  $x$ ,

$$\begin{aligned} & 1 \text{ si } x \text{ es negro y no tiene hijos rojos} \\ & 0 \text{ si } x \text{ es negro y tiene un hijo rojo} \\ & w_x / D \text{ si } x \text{ es rojo y tiene dos hijos rojos} \end{aligned}$$

Ahora redefinimos el potencial de un árbol rojo-negro  $T$  como

$$\hat{T} = \sum_{x \in T} w_x / D$$

y sea  $T''$  sea el árbol que resulta de aplicar cualquier caso no terminador de RB-INSERT-FIXUP o RB-DELETE-FIXUP a  $T$ .

F. Demuestre que  $\hat{T}'' \leq \hat{T}' + 1$  para todos los casos sin terminación de RB-INSERT FIXUP.

Argumentar que el número amortizado de modificaciones estructurales realizadas por cualquier llamada de RB-INSERT-FIXUP es  $O(1)$ .

gramo. Demuestre que  $\hat{T}'' \leq \hat{T}' + 1$  para todos los casos sin terminación de RB-DELETE FIXUP.

Argumentar que el número amortizado de modificaciones estructurales realizadas por cualquier llamada de RB-DELETE-FIXUP es  $O(1)$ .

H. Complete la demostración de que, en el peor de los casos, cualquier secuencia de  $m$  operaciones RB-INSERT y RB-DELETE realiza modificaciones estructurales  $O(m)$ .

17-5 Análisis competitivo de listas autoorganizadas con movimiento al frente Una lista autoorganizada es una lista enlazada de  $n$  elementos, en la que cada elemento tiene una clave única. Cuando buscamos un elemento en la lista, se nos da una clave y queremos encontrar un elemento con esa clave.

Una lista autoorganizada tiene dos propiedades importantes:

1. Para encontrar un elemento en la lista, dada su clave, debemos recorrer la lista desde el principio hasta encontrar el elemento con la clave dada. Si ese elemento es el elemento  $k$ -ésimo desde el principio de la lista, entonces el costo para encontrar el elemento es  $k$ .
2. Podemos reordenar los elementos de la lista después de cualquier operación, de acuerdo con una regla dada con un costo dado. Podemos elegir cualquier heurística que queramos para decidir cómo reordenar la lista.

Supongamos que comenzamos con una lista dada de  $n$  elementos y se nos da una secuencia de acceso  $D = h_1; h_2; \dots; h_m$  de claves para encontrar, en orden. El coste de la secuencia es la suma de los costes de los accesos individuales de la secuencia.

De las diversas formas posibles de reordenar la lista después de una operación, este problema se centra en la transposición de elementos de la lista adyacentes (cambiando sus posiciones en la lista) con un costo unitario para cada operación de transposición. Mostrará, por medio de una función potencial, que una heurística particular para reordenar la lista, mover al frente, implica un costo total no peor que 4 veces el de cualquier otra heurística para mantener el orden de la lista, incluso si la otra ¡La heurística conoce la secuencia de acceso de antemano!

Llamamos a este tipo de análisis un análisis competitivo.

Para una heurística  $H$  y un orden inicial dado de la lista, denote el costo de acceso de la secuencia por  $CH \cdot D$ . Sea  $m$  el número de accesos en

- a. Argumente que si la heurística  $H$  no conoce la secuencia de acceso de antemano, entonces el costo en el peor de los casos para  $H$  en una secuencia de acceso es  $CH \cdot D \cdot mn$ .

Con la heurística de mover al frente, inmediatamente después de buscar un elemento  $x$ , movemos  $x$  a la primera posición en la lista (es decir, al frente de la lista).

Sea  $\text{rank}_L(x)$  el rango del elemento  $x$  en la lista  $L$ , es decir, la posición de  $x$  en la lista  $L$ . Por ejemplo, si  $x$  es el cuarto elemento en  $L$ , entonces  $\text{rank}_L(x) = 4$ . Sea  $c_i$  el costo de acceso  $i$  usando la heurística de mover al frente, que incluye el costo de encontrar el elemento en la lista y el costo de moverlo al frente de la lista mediante una serie de transposiciones de elementos de lista adyacentes.

- b. Muestre que si  $i$  accede al elemento  $x$  en la lista  $L$  usando la heurística de mover al frente, entonces  $c_i \leq D + 2 \text{rank}_L(x) - 1$ .

Ahora comparamos move-to-front con cualquier otra heurística  $H$  que procese una secuencia de acceso de acuerdo con las dos propiedades anteriores. La heurística  $H$  puede transponer

elementos de la lista de la forma que desee, e incluso podría conocer la secuencia de acceso completa de antemano.

Sea  $L_i$  la lista después de acceder a  $i$  usando mover al frente, y sea  $L_{i+1}$  ser la lista después acceda  $i$  usando la heurística  $H$ . Denotamos el costo de acceso  $i$  por  $c_i$  para mover al frente y por  $c$  para la heurística  $H$ . Supongamos que la heurística  $H$  realiza  $t$  transposiciones durante el acceso  $i$ . En la parte (b), demostrate que

$c_i \leq rank(L_{i+1}) - rank(L_i) + 1$ . Ahora demuestra que  $c_i \leq C_t$

$$\text{rango}_{L_{i+1}} = \text{rango}_{L_i} + t$$

Definimos una inversión en la lista  $L_i$  como un par de elementos  $y$  y  $y'$  tal que  $y$  precede a  $y'$  en  $L_i$  y  $y'$  precede a  $y$  en la lista  $L_{i+1}$ . Supongamos que la lista  $L_i$  tiene inversiones de  $q_i$  después de procesar la secuencia de acceso  $h_1; h_2; \dots; h_i$ . Luego, definimos una función potencial  $\hat{c}$  que mapea  $L_i$  a un número real por  $\hat{c}(L_i) = D q_i$ . Por ejemplo, si  $L_i$  tiene los elementos  $\{e, C, a, d, b\}$  y  $L_{i+1}$  tiene los elementos  $\{h, c, b, d, e\}$ , entonces  $L_i$  tiene 5 inversiones ( $(e, c)$ ;  $(e, a)$ ;  $(e, d)$ ;  $(d, b)$ ;  $(b, e)$ ), por lo que  $\hat{c}(L_i) = 10$ . Observa que  $\hat{c}(L_{i+1}) = 0$  para todo  $i$  y que, si mover al frente y la heurística  $H$  comienzan con la misma lista  $L_0$ , entonces  $\hat{c}(L_0) = 0$ . Argumente que una transposición aumenta el potencial en 2 o disminuye el

potencial por 2.

Supongamos que  $access[i]$  encuentra el elemento  $x$ . Para comprender cómo cambia el potencial debido a  $i$ , dividamos los elementos que no sean  $x$  en cuatro conjuntos, dependiendo de dónde se encuentren en las listas justo antes del  $i$ -ésimo acceso:

El conjunto A consta de elementos que preceden a  $x$  tanto en  $L_{i+1}$  como en  $L_i$ .

El conjunto B consta de elementos que preceden a  $x$  en  $L_i$  y siguen a  $x$  en  $L_{i+1}$ .

El conjunto C consta de elementos que siguen a  $x$  en  $L_i$  y preceden a  $x$  en  $L_{i+1}$ .

El conjunto D consta de elementos que siguen a  $x$  tanto en  $L_{i+1}$  como en  $L_i$ .

mi. Argumente que  $rank(L_{i+1}) - rank(L_i) = |B| + |C|$ .

F. Demuestre que el acceso  $i$  provoca un cambio en el potencial de

$\hat{c}(L_i) - \hat{c}(L_{i+1}) = |B| + |C|$

donde, como antes, la heurística  $H$  realiza  $t$  transposiciones durante el acceso  $i$ .

Defina el costo amortizado  $cyi$  del acceso  $i$  por  $cyi = D c_i + \hat{c}(L_i) - \hat{c}(L_{i+1})$ .

gramo. Muestre que el costo amortizado  $cyi$  del acceso  $i$  está acotado superiormente por  $4c_i + 2t$ .

H. Concluya que el costo CMTF. / de secuencia de acceso con movimiento al frente es como máximo 4 veces el costo CH. / de con cualquier otra heurística  $H$ , asumiendo que ambas heurísticas comienzan con la misma lista.

### Notas del capítulo

Aho, Hopcroft y Ullman [5] utilizaron el análisis agregado para determinar el tiempo de ejecución de las operaciones en un bosque disjunto; analizaremos esta estructura de datos usando el método potencial en el Capítulo 21. Tarjan [331] analiza los métodos contables y potenciales del análisis amortizado y presenta varias aplicaciones. Atribuye el método contable a varios autores, entre ellos MR Brown, RE

Tarjan, S. Huddleston y K. Mehlhorn. Él atribuye el método potencial a DD Sleator. El término "amortizado" se debe a DD Sleator y RE Tarjan.

Las funciones potenciales también son útiles para probar los límites inferiores de ciertos tipos de problemas. Para cada configuración del problema, definimos una función potencial que asigna la configuración a un número real. Luego determinamos el potencial  $\hat{\text{init}}$  de la configuración inicial, el potencial  $\hat{\text{final}}$  de la configuración final y el cambio máximo en el potencial  $\hat{\text{max}}$  debido a cualquier paso. Por lo tanto, el número de pasos debe ser al menos  $j\hat{\text{final}} - \hat{\text{init}}_j = j\hat{\text{max}}_j$ . En los trabajos de Cormen, Sundquist y Wisniewski [79] aparecen ejemplos de funciones potenciales para demostrar límites inferiores en la complejidad de E/S; Floyd [107]; y Aggarwal y Vitter [3]. Krumme, Cybenko y Venkataraman [221] aplicaron funciones potenciales para demostrar los límites inferiores de los chismes: comunicar un elemento único de cada vértice de un gráfico a todos los demás vértices.

La heurística de pasar al frente del problema 17-5 funciona bastante bien en la práctica. Además, si reconocemos que cuando encontramos un elemento, podemos separarlo de su posición en la lista y reubicarlo al frente de la lista en tiempo constante, podemos demostrar que el costo de mover al frente es de casi el doble del costo de cualquier otra heurística, incluida, nuevamente, una que conoce la secuencia de acceso completa por adelantado.



---

## V Estructuras de datos avanzadas

---

## Introducción

Esta parte vuelve a estudiar las estructuras de datos que respaldan las operaciones en conjuntos dinámicos, pero a un nivel más avanzado que la Parte III. Dos de los capítulos, por ejemplo, hacen un uso extensivo de las técnicas de análisis amortizado que vimos en el Capítulo 17.

El Capítulo 18 presenta árboles B, que son árboles de búsqueda equilibrados diseñados específicamente para ser almacenados en discos. Debido a que los discos funcionan mucho más lentamente que la memoria de acceso aleatorio, medimos el rendimiento de los árboles B no solo por cuánto tiempo de computación consumen las operaciones de conjuntos dinámicos, sino también por cuántos accesos al disco realizan. Para cada operación de árbol B, el número de accesos al disco aumenta con la altura del árbol B, pero las operaciones de árbol B mantienen la altura baja.

El Capítulo 19 proporciona una implementación de un montón fusionable, que admite el operaciones **INSERT**, **MINIMUM**, **EXTRACT-MIN** y **UNION**.<sup>1</sup> La operación **UNION** une, o fusiona, dos montones. Los montones de Fibonacci, la estructura de datos del capítulo 19, también admiten las operaciones **DELETE** y **DECREASE-KEY**. Usamos límites de tiempo amortizados para medir el rendimiento de los montones de Fibonacci. Las operaciones **INSERT**, **MINIMUM** y **UNION** toman solo O.1/ tiempo real y amortizado en los montones de Fibonacci, y las operaciones **EXTRACT-MIN** y **DELETE** toman O.lg n/ tiempo amortizado. La ventaja más significativa de los montones de Fibonacci, sin embargo, es que **DECREASE-KEY** toma solo 0.1/ tiempo amortizado. Porque la DISMINUCION

---

<sup>1</sup>Al igual que en el problema 10-2, hemos definido un almacenamiento dinámico fusionable para admitir **MINIMUM** y **EXTRACT-MIN**, por lo que también podemos referirnos a él como un almacenamiento dinámico fusionable. Alternativamente, si fuera compatible con **MAXIMUM** y **EXTRACT-MAX**, sería un montón máximo fusionable. A menos que especifiquemos lo contrario, los montones combinables serán montones mínimos combinables de forma predeterminada.

La operación KEY toma un tiempo amortizado constante, los montones de Fibonacci son componentes clave de algunos de los algoritmos asintóticamente más rápidos hasta la fecha para problemas de gráficos.

Teniendo en cuenta que podemos superar el límite inferior  $n \lg n$  para ordenar cuando las claves son números enteros en un rango restringido, el Capítulo 20 pregunta si podemos diseñar una estructura de datos que admita las operaciones de conjunto dinámico BUSCAR, INSERTAR, ELIMINAR, MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR en  $O(\lg n)$  tiempo cuando las claves son números enteros en un rango restringido. La respuesta resulta ser que podemos, mediante el uso de una estructura de datos recursiva conocida como árbol de van Emde Boas. Si las claves son enteros únicos extraídos del conjunto  $\{0; 1; 2; \dots; u\}$ , donde  $u$  es una potencia exacta de 2, entonces los árboles de van Emde Boas soportan cada una de las operaciones anteriores en  $O(\lg \lg u)$  tiempo.

Finalmente, el Capítulo 21 presenta estructuras de datos para conjuntos disjuntos. Tenemos un universo de  $n$  elementos que se dividen en conjuntos dinámicos. Inicialmente, cada elemento pertenece a su propio conjunto singleton. La operación UNION une dos conjuntos, y la consulta FIND SET identifica el conjunto único que contiene un elemento dado en ese momento. Al representar cada conjunto como un árbol con raíz simple, obtenemos operaciones sorprendentemente rápidas: una secuencia de  $m$  operaciones se ejecuta en el tiempo  $O(m \cdot n^{\epsilon})$ , donde  $n^{\epsilon}$  es una función de crecimiento increíblemente lento— $n^{\epsilon}$  es como máximo 4 en cualquier aplicación conceible. El análisis amortizado que prueba este límite de tiempo es tan complejo como simple es la estructura de datos.

Los temas cubiertos en esta parte no son de ninguna manera los únicos ejemplos de "avanzado" estructuras de datos. Otras estructuras de datos avanzadas incluyen lo siguiente:

Los árboles dinámicos, presentados por Sleator y Tarjan [319] y discutidos por Tarjan [330], mantienen un bosque de árboles con raíces desarticuladas. Cada arista en cada árbol tiene un costo de valor real. Los árboles dinámicos admiten consultas para encontrar padres, raíces, costos de borde y el costo de borde mínimo en una ruta simple desde un nodo hasta una raíz.

Los árboles se pueden manipular cortando bordes, actualizando todos los costos de borde en una ruta simple desde un nodo hasta una raíz, vinculando una raíz a otro árbol y haciendo que un nodo sea la raíz del árbol en el que aparece. Una implementación de árboles dinámicos da un  $O(\lg n)$  límite de tiempo amortizado para cada operación; una implementación más complicada produce  $O(\lg n)$  límites de tiempo en el peor de los casos. Los árboles dinámicos se utilizan en algunos de los algoritmos de flujo de red asintóticamente más rápidos.

Los árboles splay, desarrollados por Sleator y Tarjan [320] y, nuevamente, discutidos por Tarjan [330], son una forma de árbol de búsqueda binaria en el que las operaciones estándar del árbol de búsqueda se ejecutan en  $O(\lg n)$  tiempo amortizado. Una aplicación de árboles splay simplifica los árboles dinámicos.

Las estructuras de datos persistentes permiten consultas y, a veces, también actualizaciones, en versiones anteriores de una estructura de datos. Driscoll, Sarnak, Sleator y Tarjan [97] presentan técnicas para hacer que las estructuras de datos enlazadas sean persistentes con solo un pequeño período de tiempo.

y costo de espacio. El problema 13-1 da un ejemplo simple de un conjunto dinámico persistente.

Como en el Capítulo 20, varias estructuras de datos permiten una implementación más rápida de las operaciones del diccionario (INSERTAR, ELIMINAR y BÚSQUEDA) para un universo restringido de claves. Al aprovechar estas restricciones, pueden lograr mejores tiempos de ejecución asintóticos en el peor de los casos que las estructuras de datos basadas en la comparación.

Fredman y Willard introdujeron los árboles de fusión [115], que fueron la primera estructura de datos que permitió operaciones de diccionario más rápidas cuando el universo está restringido a números enteros. Mostraron cómo implementar estas operaciones en tiempo  $O.\lg n = \lg \lg n/$ . Varias estructuras de datos posteriores, incluidos los árboles de búsqueda exponencial [16], también han proporcionado límites mejorados en algunas o todas las operaciones del diccionario y se mencionan en las notas de los capítulos a lo largo de este libro.

Las estructuras de datos de gráficos dinámicos admiten varias consultas al tiempo que permiten que la estructura de un gráfico cambie a través de operaciones que insertan o eliminan vértices o bordes. Los ejemplos de las consultas que admiten incluyen conectividad de vértice [166], conectividad de borde, árboles de expansión mínimos [165], biconectividad y cierre transitivo [164].

Las notas de los capítulos a lo largo de este libro mencionan estructuras de datos adicionales.

Los árboles B son árboles de búsqueda equilibrados diseñados para funcionar bien en discos u otros dispositivos de almacenamiento secundario de acceso directo. Los árboles B son similares a los árboles rojo-negro (Capítulo 13), pero son mejores para minimizar las operaciones de E/S del disco. Muchos sistemas de bases de datos utilizan árboles B, o variantes de árboles B, para almacenar información.

Los árboles B se diferencian de los árboles rojo-negro en que los nodos del árbol B pueden tener muchos hijos, desde unos pocos hasta miles. Es decir, el "factor de ramificación" de un árbol B puede ser bastante grande, aunque normalmente depende de las características de la unidad de disco utilizada. Los árboles B son similares a los árboles rojo-negro en que cada árbol B de  $n$  nodos tiene una altura  $O.\lg n/$ . Sin embargo, la altura exacta de un árbol B puede ser considerablemente menor que la de un árbol rojo-negro porque su factor de ramificación  $y$ , por lo tanto, la base del logaritmo que expresa su altura, puede ser mucho mayor. Por lo tanto, también podemos usar árboles B para implementar muchas operaciones de conjuntos dinámicos en el tiempo  $O.\lg n/$ .

Los árboles B generalizan los árboles de búsqueda binarios de forma natural. La figura 18.1 muestra un árbol B simple. Si un nodo de árbol B interno  $x$  contiene claves  $x:n$ , entonces  $x$  tiene  $x:n C 1$  hijos. Las claves en el nodo  $x$  sirven como puntos de división que separan el rango de claves manejadas por  $x$  en subrangos  $x:n C 1$ , cada uno manejado por un hijo de  $x$ . Al buscar una clave en un árbol B, tomamos una decisión  $x:n C 1/way$  basada en comparaciones con las claves  $x:n$  almacenadas en el nodo  $x$ . La estructura de los nodos hoja difiere de la de los nodos internos; examinaremos estas diferencias en la Sección 18.1.

La sección 18.1 da una definición precisa de árboles B y prueba que la altura de un árbol B crece solo logarítmicamente con el número de nodos que contiene. La Sección 18.2 describe cómo buscar una clave e insertar una clave en un árbol B, y la Sección 18.3 analiza la eliminación. Sin embargo, antes de continuar, debemos preguntarnos por qué evaluamos las estructuras de datos diseñadas para funcionar en un disco de manera diferente a las estructuras de datos diseñadas para funcionar en la memoria principal de acceso aleatorio.

#### Estructuras de datos en almacenamiento secundario

Los sistemas informáticos aprovechan diversas tecnologías que proporcionan capacidad de memoria. La memoria primaria (o memoria principal) de un sistema informático normalmente

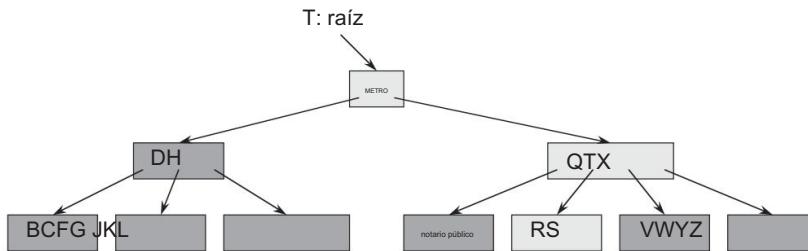


Figura 18.1 Un árbol B cuyas claves son las consonantes del inglés. Un nodo interno  $x$  que contiene claves  $x:n$  tiene hijos  $x:n$ . Todas las hojas están a la misma profundidad en el árbol. Los nodos ligeramente sombreados se examinan en busca de la letra R.

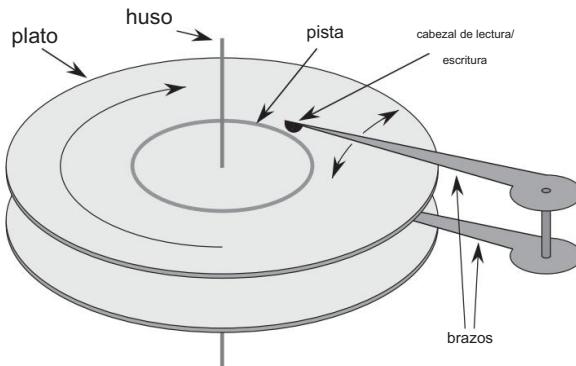


Figura 18.2 Una unidad de disco típica. Comprende uno o más platos (aquí se muestran dos platos) que giran alrededor de un eje. Cada plato se lee y escribe con una cabeza al final de un brazo. Los brazos giran alrededor de un eje de pivote común. Una pista es la superficie que pasa por debajo del cabezal de lectura/escritura cuando el cabezal está estacionario.

consta de chips de memoria de silicio. Esta tecnología suele ser un orden de magnitud más cara por bit almacenado que la tecnología de almacenamiento magnético, como cintas o discos. La mayoría de los sistemas informáticos también tienen almacenamiento secundario basado en discos magnéticos; la cantidad de dicho almacenamiento secundario a menudo supera la cantidad de memoria primaria en al menos dos órdenes de magnitud.

La figura 18.2 muestra una unidad de disco típica. El accionamiento consta de uno o más platos, que giran a una velocidad constante alrededor de un eje común. Un material magnetizable cubre la superficie de cada plato. La unidad lee y escribe cada plato por medio de una cabeza al final de un brazo. Los brazos pueden mover la cabeza hacia o desde

del husillo. Cuando un cabezal dado está estacionario, la superficie que pasa por debajo se llama pista . Varios platos aumentan solo la capacidad de la unidad de disco y no su rendimiento.

Aunque los discos son más baratos y tienen mayor capacidad que la memoria principal, son mucho, mucho más lentos porque tienen partes mecánicas móviles.<sup>1</sup> El movimiento mecánico tiene dos componentes: la rotación del plato y el movimiento del brazo. En el momento de escribir este artículo, los discos básicos giran a velocidades de 5400 a 15 000 revoluciones por minuto (RPM).

Por lo general, vemos velocidades de 15 000 RPM en unidades de servidor, velocidades de 7200 RPM en unidades para computadoras de escritorio y velocidades de 5400 RPM en unidades para computadoras portátiles. Aunque 7200 RPM puede parecer rápido, una rotación tarda 8,33 milisegundos, que es más de 5 órdenes de magnitud más que los tiempos de acceso de 50 nanosegundos (más o menos) que se encuentran comúnmente para la memoria de silicio. En otras palabras, si tenemos que esperar una rotación completa para que un elemento en particular pase al cabezal de lectura/escritura, podríamos acceder a la memoria principal más de 100 000 veces durante ese lapso. En promedio, tenemos que esperar solo la mitad de una rotación, pero aún así, la diferencia en los tiempos de acceso para la memoria de silicio en comparación con los discos es enorme. Mover los brazos también lleva algo de tiempo. En el momento de escribir este artículo, los tiempos de acceso promedio para los discos básicos están en el rango de 8 a 11 milisegundos.

Para amortizar el tiempo de espera de los movimientos mecánicos, los discos acceden no solo a un elemento sino a varios a la vez. La información se divide en un número de páginas de bits de igual tamaño que aparecen consecutivamente dentro de las pistas, y cada disco leído o escrito es de una o más páginas completas. Para un disco típico, una página puede tener una longitud de 211 a 214 bytes. Una vez que el cabezal de lectura/escritura está colocado correctamente y el disco ha girado hasta el comienzo de la página deseada, la lectura o escritura de un disco magnético es completamente electrónica (aparte de la rotación del disco), y el disco puede leer o escribir rápidamente grandes cantidades de datos.

A menudo, acceder a una página de información y leerla desde un disco lleva más tiempo que examinar toda la información leída. Por esta razón, en este capítulo examinaremos por separado los dos componentes principales del tiempo de ejecución:

el número de accesos al disco, y

el tiempo de CPU (computación).

Medimos la cantidad de accesos al disco en términos de la cantidad de páginas de información que deben leerse o escribirse en el disco. Observamos que el tiempo de acceso al disco no es constante: depende de la distancia entre la pista actual y la pista deseada y también de la posición de rotación inicial del disco. Deberíamos

---

<sup>1</sup>En el momento de escribir este artículo, las unidades de estado sólido han llegado recientemente al mercado de consumo. Aunque son más rápidos que las unidades de disco mecánicas, cuestan más por gigabyte y tienen capacidades más bajas que las unidades de disco mecánicas.

no obstante, utilice el número de páginas leídas o escritas como una aproximación de primer orden del tiempo total empleado en acceder al disco.

En una aplicación típica de árbol B, la cantidad de datos manejados es tan grande que no todos los datos caben en la memoria principal a la vez. Los algoritmos B-tree copian las páginas seleccionadas del disco a la memoria principal según sea necesario y vuelven a escribir en el disco las páginas que han cambiado. Los algoritmos de árbol B mantienen solo un número constante de páginas en la memoria principal en cualquier momento; por lo tanto, el tamaño de la memoria principal no limita el tamaño de los árboles B que se pueden manejar.

Modelamos las operaciones de disco en nuestro pseudocódigo de la siguiente manera. Sea  $x$  un puntero a un objeto. Si el objeto está actualmente en la memoria principal de la computadora, entonces podemos referirnos a los atributos del objeto como de costumbre:  $x$ : clave, por ejemplo. Sin embargo, si el objeto al que se refiere  $x$  reside en el disco, entonces debemos realizar la operación DISK-READ. $x/$  para leer el objeto  $x$  en la memoria principal antes de que podamos referirnos a sus atributos. (Suponemos que si  $x$  ya está en la memoria principal, entonces DISK-READ. $x/$  no requiere acceso al disco; no es operativo). De manera similar, la operación DISK-WRITE. $x/$  se usa para guardar cualquier cambios que se han hecho a los atributos del objeto  $x$ . Es decir, el patrón típico para trabajar con un objeto es el siguiente:

```
x D un puntero a algún objeto DISK-
READ.x/ operaciones
que acceden y/o modifican los atributos de x DISK-WRITE.x/ // se
omite si no se cambiaron los atributos de x otras operaciones que acceden pero no modifican
los atributos de X
```

El sistema puede mantener solo un número limitado de páginas en la memoria principal en un momento dado. Supondremos que el sistema se vacía de las páginas de la memoria principal que ya no están en uso; nuestros algoritmos de árbol B ignorarán este problema.

Dado que en la mayoría de los sistemas el tiempo de ejecución de un algoritmo de árbol B depende principalmente del número de operaciones DISK-READ y DISK-WRITE que realiza, por lo general queremos que cada una de estas operaciones lea o escriba la mayor cantidad de información posible. Por lo tanto, un nodo de árbol B suele ser tan grande como una página de disco completa, y este tamaño limita la cantidad de elementos secundarios que puede tener un nodo de árbol B.

Para un árbol B grande almacenado en un disco, a menudo vemos factores de ramificación entre 50 y 2000, según el tamaño de una clave en relación con el tamaño de una página. Un factor de ramificación grande reduce drásticamente tanto la altura del árbol como la cantidad de accesos al disco necesarios para encontrar cualquier clave. La Figura 18.3 muestra un árbol B con un factor de ramificación de 1001 y una altura de 2 que puede almacenar más de mil millones de claves; sin embargo, dado que podemos mantener el nodo raíz permanentemente en la memoria principal, podemos encontrar cualquier clave en este árbol haciendo como máximo dos accesos al disco.

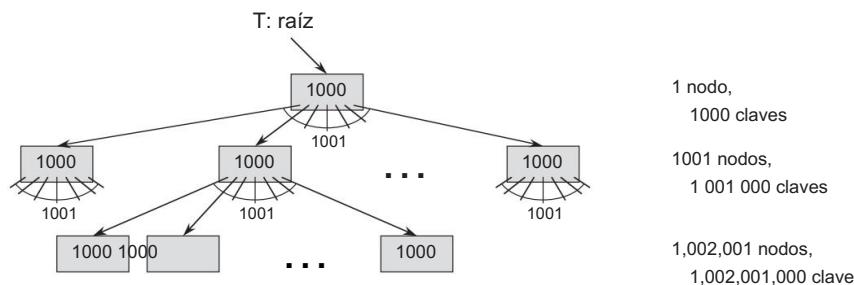


Figura 18.3 Un árbol B de altura 2 que contiene más de mil millones de claves. Dentro de cada nodo  $x$  se muestra  $x:n$ , el número de claves en  $x$ . Cada nodo interno y hoja contiene 1000 claves. Este árbol B tiene 1001 nodos en la profundidad 1 y más de un millón de hojas en la profundidad 2.

## 18.1 Definición de árboles B

Para mantener las cosas simples, asumimos, como lo hemos hecho para los árboles de búsqueda binarios y los árboles rojo-negro, que cualquier "información satelital" asociada con una clave reside en el mismo nodo que la clave. En la práctica, uno podría almacenar con cada clave solo un puntero a otra página de disco que contenga la información del satélite para esa clave. El pseudocódigo de este capítulo asume implícitamente que la información del satélite asociada con una clave, o el puntero a dicha información del satélite, viaja con la clave cada vez que la clave se mueve de un nodo a otro. Una variante común de un árbol B, conocida como árbol BC, almacena toda la información del satélite en las hojas y almacena solo claves y punteros secundarios en los nodos internos, maximizando así el factor de ramificación de los nodos internos.

Un árbol B T es un árbol con raíz (cuya raíz es T:raíz) que tiene las siguientes propiedades:

1. Cada nodo x tiene los siguientes atributos:
    - a. x:n, el número de claves almacenadas actualmente en el nodo x, b. las propias claves x:n , x:clave1;x:clave2;:::;x:clavex: n, almacenadas en orden no decreciente, de modo que x:clave1 x:clave2 c. x:hoja , un valor booleano que es VERDADERO si x es una hoja y FALSO si x es un nodo interno.
  2. Cada nodo interno x también contiene x:n C 1 punteros x:c1;x:c2;:::;x:c<sub>x</sub>: nC1 a sus hijos Los nodos hoja no tienen hijos, por lo que sus atributos ci no están definidos.

3. Las claves  $x:\text{key}_i$  separan los rangos de claves almacenadas en cada subárbol: si  $k_i$  es cualquiera clave almacenada en el subárbol con raíz  $x:c_i$ , luego

$k_1 \leq \text{tecla}_1 \leq k_2 \leq \text{tecla}_2$

$x: \text{tecla } x: n \quad kx: nC1 :$

4. Todas las hojas tienen la misma profundidad, que es la altura del árbol  $h$ .

5. Los nodos tienen límites inferiores y superiores en el número de claves que pueden contener.

Expresamos estos límites en términos de un número entero fijo  $t \geq 2$  llamado grado mínimo del árbol  $B$ :

- a. Cada nodo que no sea la raíz debe tener al menos  $t - 1$  claves. Cada nodo interno que no sea la raíz tiene por lo menos  $t$  hijos. Si el árbol no está vacío, la raíz debe tener al menos una clave.
- b. Cada nodo puede contener como máximo

$2t - 1$  claves. Por lo tanto, un nodo interno puede tener como máximo  $2t$  hijos. Decimos que un nodo está lleno si contiene exactamente  $2t - 1$  claves.<sup>2</sup>

El árbol  $B$  más simple ocurre cuando  $t = 2$ . Entonces, cada nodo interno tiene 2, 3 o 4 hijos, y tenemos un árbol 2-3-4. En la práctica, sin embargo, valores mucho mayores de  $t$  producen árboles  $B$  de menor altura.

#### La altura de un árbol $B$

El número de accesos al disco necesarios para la mayoría de las operaciones en un árbol  $B$  es proporcional a la altura del árbol  $B$ . Ahora analizamos la altura del peor de los casos de un árbol  $B$ .

#### Teorema 18.1

Si  $n \geq 1$ , entonces para cualquier árbol  $B$   $T$  de clave  $n$  de altura  $h$  y grado mínimo  $t \geq 2$ ,

$$h \geq \frac{n - 1}{2^t - 1}$$

Prueba La raíz de un árbol  $B$   $T$  contiene al menos una clave, y todos los demás nodos contienen al menos  $t - 1$  claves. Así,  $T$  cuya altura es  $h$ , tiene al menos 2 nodos en la profundidad 1, al menos  $2t$  nodos en la profundidad 2, al menos  $2t^2$  nodos<sup>2</sup> en la profundidad 3, y así sucesivamente, hasta que en la profundidad  $h$  tenga al menos  $2th - 1$  nodos. La figura 18.4 ilustra tal árbol para  $h = 3$ . Por lo tanto, el

---

<sup>2</sup>Otra variante común en un árbol  $B$ , conocida como árbol  $B$ , requiere que cada nodo interno esté lleno al menos  $2t - 1$ , en lugar de al menos la mitad, como lo requiere un árbol  $B$ .

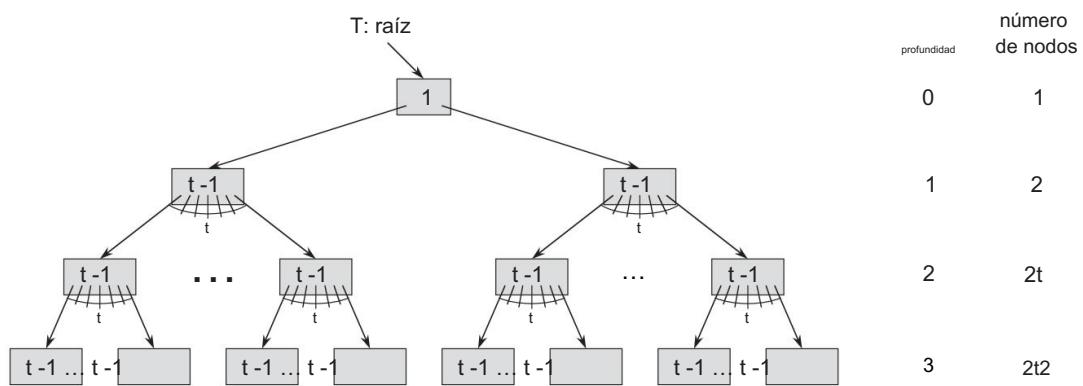


Figura 18.4 Un árbol B de altura 3 que contiene el mínimo número posible de claves. Dentro de cada nodo x se muestra  $x:n$ .

número n de claves satisface la desigualdad

$$\begin{aligned}
 & h \\
 & \text{notas} \quad 1 C .t 1/X \quad 2ti1 \\
 & \quad iD1 \\
 & D 1 C 2.t 1/th 1 \quad \frac{2ti1}{t - 1} \\
 & D 2^o 1
 \end{aligned}$$

Por álgebra simple, obtenemos  $.n C 1/2 = 2$ . Tomando logaritmos en base t de ambos lados que prueba el teorema. ■

Aquí vemos el poder de los árboles B, en comparación con los árboles rojo-negro. Aunque la altura del árbol crece como  $O.\lg n$  en ambos casos (recuerde que t es una constante), para árboles B la base del logaritmo puede ser muchas veces mayor. Por lo tanto, los árboles B ahorran un factor de aproximadamente  $\lg t$  sobre los árboles rojo-negro en el número de nodos examinados para la mayoría de las operaciones de árboles. Debido a que generalmente tenemos que acceder al disco para examinar un nodo arbitrario en un árbol, los árboles B evitan una cantidad sustancial de accesos al disco.

### Ejercicios

18.1-1

¿Por qué no permitimos un grado mínimo de  $t-1$ ?

18.1-2

¿Para qué valores de t el árbol de la figura 18.1 es un árbol B legal?

## 18.1-3

Muestre todos los árboles B legales de grado mínimo 2 que representen f1; 2; 3; 4; 5g

## 18.1-4

En función del grado mínimo t, ¿cuál es el número máximo de claves que se pueden almacenar en un árbol B de altura h?

## 18.1-5

Describa la estructura de datos que resultaría si cada nodo negro en un árbol rojo-negro iban a absorber a sus hijos rojos, incorporando a sus hijos con los suyos.

## 18.2 Operaciones básicas en árboles B

En esta sección, presentamos los detalles de las operaciones B-ÁRBOL-BÚSQUEDA, B-ÁRBOL-CREAR y B-ÁRBOL-INSERTAR. En estos procedimientos, adoptamos dos convenciones:

La raíz del árbol B siempre está en la memoria principal, por lo que nunca necesitamos realizar una LECTURA DE DISCO en la raíz; Sin embargo, tenemos que realizar una ESCRITURA EN DISCO de la raíz cada vez que se cambia el nodo raíz.

Todos los nodos que se pasan como parámetros ya deben tener una operación DISK-READ realizada en ellos.

Los procedimientos que presentamos son todos algoritmos de "una pasada" que proceden hacia abajo desde la raíz del árbol, sin tener que retroceder.

### Buscando un árbol B

Buscar en un árbol B es muy parecido a buscar en un árbol de búsqueda binario, excepto que en lugar de tomar una decisión de bifurcación binaria o "bidireccional" en cada nodo, tomamos una decisión de bifurcación multidireccional de acuerdo con el número de hijos del nodo. Más precisamente, en cada nodo interno x, tomamos una decisión de bifurcación .x:n C 1-/way.

B-TREE-SEARCH es una generalización directa del procedimiento TREE-SEARCH definido para árboles de búsqueda binarios. B-TREE-SEARCH toma como entrada un puntero al nodo raíz x de un subárbol y una clave k para buscar en ese subárbol. Por lo tanto, la llamada de nivel superior tiene la forma B-TREE-SEARCH.T:root; k/. Si k está en el árbol B, B-ÁRBOL-BÚSQUEDA devuelve el par ordenado .y; i / que consiste en un nodo y un índice i tal que y:key[i] D k. De lo contrario, el procedimiento devuelve NIL.

```
B-ÁRBOL-BÚSQUEDA.x;
k/ 1 i D 1
2 while ix: nyk > x:keyi 3 i D i C 1 4 if
ix: nyk == x:keyi 5
return .x; i / 6 elseif x:hoja 7 return
NIL 8 else DISK-
READ.x:ci/ return
B-TREE-SEARCH.x:ci;
k/ 9
```

Usando un procedimiento de búsqueda lineal, las líneas 1–3 encuentran el índice  $i$  más pequeño tal que  $k < x:\text{key}_i$ , o bien establecen  $i$  en  $x:n$ . Las líneas 4–5 verifican si ahora hemos descubierto la clave, volviendo si tenemos. De lo contrario, las líneas 6 a 9 terminan la búsqueda sin éxito (si  $x$  es una hoja) o recurren para buscar el subárbol apropiado de  $x$ , después de realizar la LECTURA DE DISCO necesaria en ese hijo.

La figura 18.1 ilustra el funcionamiento de B-TREE-SEARCH. El examen de procedimiento ines los nodos ligeramente sombreados durante una búsqueda de la clave  $R$ .

Como en el procedimiento TREE-SEARCH para búsqueda de árboles binarios, los nodos encontrados durante la recursión forman un camino simple hacia abajo desde la raíz del árbol. Por lo tanto, el procedimiento B-TREE-SEARCH accede a las páginas de disco  $O(h/D \log n)$ , donde  $h$  es la altura del árbol  $B$  y  $n$  es el número de claves en el árbol  $B$ .

Dado que  $x:n < 2t$ , el ciclo while de las líneas 2–3 toma tiempo  $O(t)$  dentro de cada nodo, y el tiempo total de CPU es  $O(th/D \log n)$ .

#### Crear un árbol B vacío

Para construir un árbol  $B[T]$ , primero usamos B-TREE-CREATE para crear un nodo raíz vacío y luego llame a B-TREE-INSERT para agregar nuevas claves. Ambos procedimientos utilizan un procedimiento auxiliar ALLOCATE-NODE, que asigna una página de disco para ser utilizada como un nuevo nodo en el tiempo  $O(1)$ . Podemos suponer que un nodo creado por ALLOCATE NODE no requiere DISK-READ, ya que todavía no hay información útil almacenada en el disco para ese nodo.

```
B-TREE-CREATE.T / 1
x D ALLOCATE-NODE./ 2 x:hoja
D VERDADERO 3 x:n
D 0
4 ESCRITURA EN DISCO.x/
5 T: raíz D x
```

B-TREE-CREATE requiere  $O(1)$  operaciones de disco y  $O(1)$  tiempo de CPU.

### Insertar una clave en un árbol B

Insertar una clave en un árbol B es significativamente más complicado que insertar una clave en un árbol de búsqueda binario. Al igual que con los árboles de búsqueda binarios, buscamos la posición de la hoja en la que insertar la nueva clave. Sin embargo, con un árbol B, no podemos simplemente crear un nuevo nodo de hoja e insertarlo, ya que el árbol resultante no sería un árbol B válido.

En su lugar, insertamos la nueva clave en un nodo hoja existente. Como no podemos insertar una clave en un nodo hoja que está lleno, introducimos una operación que divide un nodo completo y (que tiene  $2t + 1$  claves) alrededor de su clave mediana  $y: key$  en dos nodos que tienen solo  $t + 1$  claves cada uno. La clave de la mediana sube al padre de  $y$  para identificar el punto de división entre los dos árboles nuevos. Pero si el padre de  $y$  también está lleno, debemos dividirlo antes de que podamos insertar la nueva clave  $y$ , por lo tanto, podríamos terminar dividiendo los nodos completos en todo el árbol.

Al igual que con un árbol de búsqueda binaria, podemos insertar una clave en un árbol B en un solo paso por el árbol desde la raíz hasta una hoja. Para hacerlo, no esperamos a saber si realmente necesitaremos dividir un nodo completo para realizar la inserción. En cambio, a medida que viajamos por el árbol en busca de la posición a la que pertenece la nueva clave, dividimos cada nodo completo que encontramos en el camino (incluida la hoja misma). Por lo tanto, siempre que queramos dividir un nodo completo  $y$ , estamos seguros de que su padre no está completo.

### División de un nodo en un árbol B

El procedimiento B-TREE-SPLIT-CHILD toma como entrada un nodo interno  $x$  no completo (se supone que está en la memoria principal) y un índice  $i$  tal que  $x: ci$  (también se supone que está en la memoria principal).  $memoria$  es un hijo completo de  $x$ . Luego, el procedimiento divide este hijo en dos y ajusta  $x$  para que tenga un hijo adicional. Para dividir una raíz completa, primero convertiremos la raíz en un elemento secundario de un nuevo nodo raíz vacío, de modo que podamos usar B-TREE-SPLIT-CHILD. El árbol crece así en altura por uno; dividir es el único medio por el cual el árbol crece.

La figura 18.5 ilustra este proceso. Dividimos el nodo completo  $y$  sobre su clave mediana  $S$ , que sube al nodo padre  $x$  de  $y$ . Esas claves en  $y$  que son mayores que la clave mediana se mueven a un nuevo nodo  $'$ , que se convierte en un nuevo hijo de  $x$ .

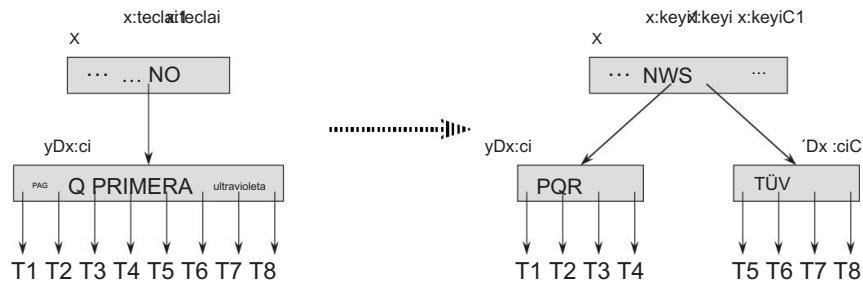


Figura 18.5 División de un nodo con  $t = 4$ . El nodo  $y D x:ci$  se divide en dos nodos,  $y$  y  $'$ , y la clave mediana S de  $y$  sube al padre de  $y$ .

#### B-ÁRBOL-SPLIT-NIÑO.x; i /

```
1 ' D ALLOCATE-NODE./ 2 y D
x:ci 3 ':hoja D
y:hoja 4 ':n D t 1 5
para j D 1 a t 6
':keyj D y:keyjCt 7      1
si no y:hoja 8 para j D 1 a t
```

```
9          ':cj D y:cjCt 10
y:n D t 1 11 para j
D x:n C 1 hasta i C 1 12 x:cjC1 D x:cj 13
x:ciC1 D ' 14 para j D x:n
hasta i 15
x:teclajC1 D x:teclaj 16 x:teclai
D y:tecla 17 x:n D x:n C 1 18
DISK-WRITE.y/ 19
DISK-WRITE./ 20
DISK-WRITE.x/
```

B-TREE-SPLIT-CHILD funciona simplemente "cortando y pegando". Aquí,  $x$  es el nodo que se está dividiendo y  $y$  es el hijo de  $x$  (establecido en la línea 2). El nodo  $y$  originalmente tiene  $2t$  hijos ( $2t - 1$  claves) pero se reduce a  $t$  hijos ( $t - 1$  claves) mediante esta operación. El nodo  $'$  toma los  $t$  hijos más grandes ( $t - 1$  claves) de  $y$ , y  $'$  se convierte en un nuevo hijo.

de x, colocado justo después de y en la tabla de niños de x. La clave mediana de y se mueve hacia arriba para convertirse en la clave en x que separa y y'.

Las líneas 1–9 crean el nodo 'y le dan el t más grande 1 claves y correspondientes t hijos de y. La línea 10 ajusta el recuento de claves para y. Finalmente, las líneas 11 a 17 insertan ' como hijo de x, mueven la tecla mediana de y hasta x para separar y de ', y ajustan el conteo de teclas de x. Las líneas 18 a 20 escriben todas las páginas de disco modificadas. El tiempo de CPU utilizado por B-TREE-SPLIT-CHILD es ,t/, debido a los bucles en las líneas 5–6 y 8–9. (Los otros bucles se ejecutan para iteraciones O/t). El procedimiento realiza operaciones de disco O.1/.

#### Inserción de una clave en un árbol B en una sola pasada por el árbol

Insertamos una clave k en un árbol B T de altura h en una sola pasada por el árbol, lo que requiere accesos de disco O(h). El tiempo de CPU requerido es O.th/D Ot logt n/. El procedimiento B-TREE-INSERT utiliza B-TREE-SPLIT-CHILD para garantizar que la recursión nunca descienda a un nodo completo.

#### B-ÁRBOL-INSERCIÓN.T; k/

```

1 r D T: raíz
2 si r:n == 2t 1
3     s D ASIGNAR-NODO./
4     T: raíz D s
5     s:hoja D FALSO s:n
6     D 0
7     s:c1 D r B-
8     ÁRBOL-DIVIDIDO-NIÑO.s; 1/ B-
9     ÁRBOL-INSERCIÓN-NO COMPLETO.s;
k/ 10 else B-ÁRBOL-INSERTAR-NO COMPLETO.r; k/

```

Las líneas 3 a 9 manejan el caso en el que el nodo raíz r está lleno: la raíz se divide y un nuevo nodo s (que tiene dos hijos) se convierte en la raíz. Dividir la raíz es la única forma de aumentar la altura de un árbol B. La figura 18.6 ilustra este caso. A diferencia de un árbol de búsqueda binaria, un árbol B aumenta en altura en la parte superior en lugar de en la parte inferior.

El procedimiento finaliza llamando a B-TREE-INSERT-NONFULL para insertar la clave k en el árbol cuya raíz es el nodo raíz no completo. B-TREE-INSERT-NOFULL se repite como necesario en el árbol, garantizando en todo momento que el nodo al que recurre no está lleno llamando a B-TREE-SPLIT-CHILD según sea necesario.

El procedimiento recursivo auxiliar B-TREE-INSERT-NOFULL inserta la clave k en el nodo x, que se supone que no está lleno cuando se llama al procedimiento. La operación de B-ÁRBOL-INSERTAR y la operación recursiva de B-ÁRBOL-INSERTAR-NO COMPLETO garantizan que esta suposición es verdadera.

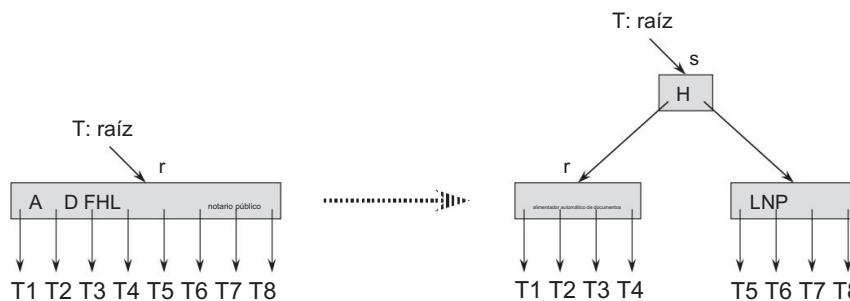


Figura 18.6 División de la raíz con  $t = 4$ . El nodo raíz  $r$  se divide en dos y se crea un nuevo nodo raíz  $s$ . La nueva raíz contiene la clave mediana de  $r$  y tiene las dos mitades de  $r$  como hijos. El árbol B crece en altura en uno cuando se parte la raíz.

#### B-ÁRBOL-INSERTAR-NO COMPLETO.x; k/

```

1 i D x: n
2 si x: hoja 3
mientras yo      1 y k < x:keyi
4           x:keyiC1 D x:keyi i D i
5           1
6           x:keyiC1 D kx:n
7           D x:n C 1 DISK-
           WRITE.x/ 8 9 else
while i      1 y k < x: clavei
10          yo re yo   1
11          i D i C 1
12          DISK-READ.x:ci/ if
13          x:ci:n == 2t 1 B-TREE-
           SPLIT-CHILD.x; i / si k > x:keyi i D
15          i C 1 B-ÁRBOL-
           INSERTAR-
17          NO COMPLETO.x:ci; k/

```

El procedimiento B-TREE-INSERT-NOFULL funciona de la siguiente manera. Las líneas 3 a 8 manejan el caso en el que  $x$  es un nodo hoja al insertar la clave  $k$  en  $x$ . Si  $x$  no es un nodo hoja, entonces debemos insertar  $k$  en el nodo hoja apropiado en el subárbol con raíz en el nodo interno  $x$ . En este caso, las líneas 9 a 11 determinan el hijo de  $x$  al que desciende la recursividad. La línea 13 detecta si la recursividad descendería a un máximo niño, en cuyo caso la línea 14 usa B-ÁRBOL-DIVIDIDO-NIÑO para dividir ese niño en dos niños no completos, y las líneas 15 y 16 determinan cuál de los dos niños es ahora el

correcta para descender. (Tenga en cuenta que no es necesario un DISK-READ.x:ci/ después de que la línea 16 incremente i, ya que la recursividad descenderá en este caso a un hijo que acaba de crear B-TREE-SPLIT-CHILD). La red El efecto de las líneas 13 a 16 es, por lo tanto, garantizar que el procedimiento nunca recurrira a un nodo completo. La línea 17 luego recurre para insertar k en el subárbol apropiado. La figura 18.7 ilustra los diversos casos de inserción en un árbol B.

Para un árbol B de altura h, B-TREE-INSERT realiza accesos de disco O<sub>h</sub>, ya que solo se producen operaciones O.1/ DISK-READ y DISK-WRITE entre llamadas a B-TREE-INSERT-NONFULL. El tiempo de CPU total utilizado es O.th/ D Ot logt n/.

Dado que B-TREE-INSERT-NONFULL es recursivo en la cola, podemos implementarlo alternativamente como un ciclo while , demostrando así que el número de páginas que deben estar en la memoria principal en cualquier momento es O.1/.

### Ejercicios

#### 18.2-1

Muestre los resultados de insertar las teclas

F; S; P; K; C; L; H; T; V; W; METRO; R; NORTE; PAG; A; B; X; Y; D; Z; E

en orden en un árbol B vacío con un grado mínimo de 2. Dibuje solo las configuraciones del árbol justo antes de que algún nodo deba dividirse, y también dibuje la configuración final.

#### 18.2-2

Explique bajo qué circunstancias, si las hay, se producen operaciones redundantes DISK-READ o DISK-WRITE durante el curso de la ejecución de una llamada a B-TREE-INSERT. (Una LECTURA DE DISCO redundante es una LECTURA DE DISCO para una página que ya está en la memoria. Un DISK-WRITE redundante escribe en el disco una página de información que es idéntica a la que ya está almacenada allí).

#### 18.2-3

Explique cómo encontrar la clave mínima almacenada en un árbol B y cómo encontrar el predecesor de una clave dada almacenada en un árbol B.

#### 18.2-4 ?

Supongamos que insertamos las teclas f1; 2; : : : ; ng en un árbol B vacío con un grado mínimo de 2. ¿Cuántos nodos tiene el árbol B final?

#### 18.2-5

Dado que los nodos hoja no requieren punteros a los hijos, posiblemente podrían usar un valor t diferente (mayor) que los nodos internos para el mismo tamaño de página de disco. Muestre cómo modificar los procedimientos para crear e insertar en un árbol B para manejar esta variación.

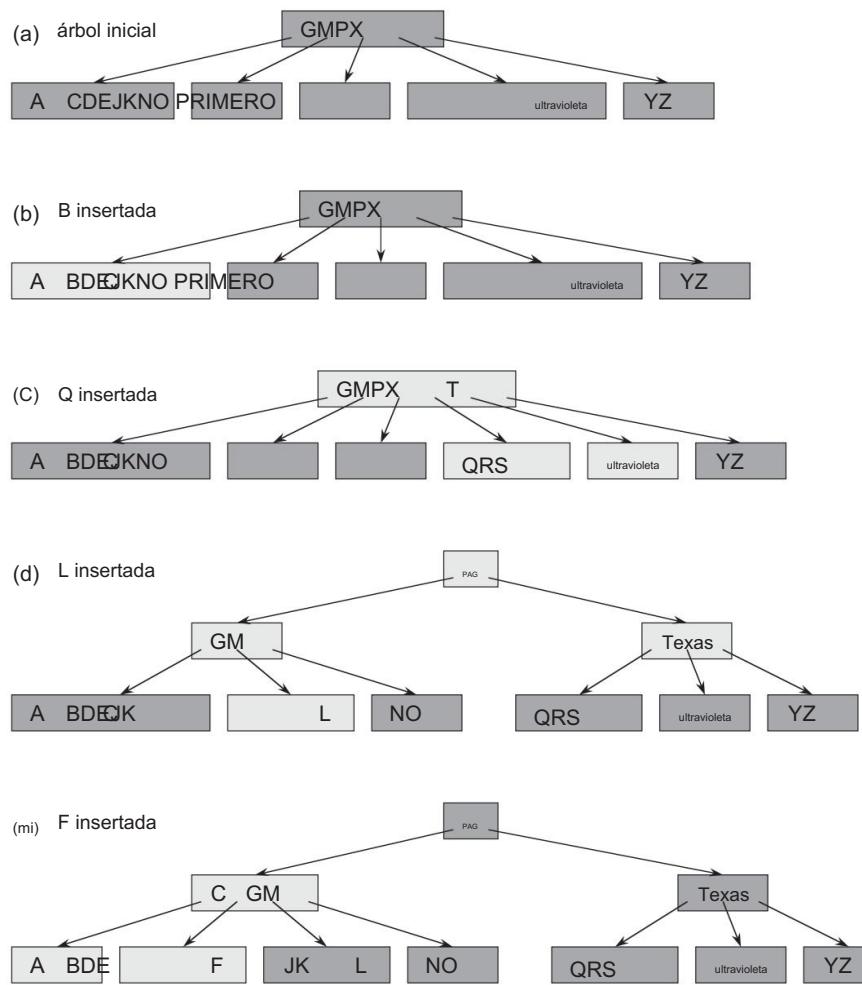


Figura 18.7 Inserción de claves en un árbol B. El grado mínimo  $t$  para este árbol B es 3, por lo que un nodo puede contener como máximo 5 claves. Los nodos modificados por el proceso de inserción están ligeramente sombreados. (a) El árbol inicial para este ejemplo. (b) El resultado de insertar B en el árbol inicial; esta es una simple inserción en un nodo hoja. (c) El resultado de insertar Q en el árbol anterior. El nodo RSTUV se divide en dos nodos que contienen RS y UV, la tecla T se mueve hacia la raíz y Q se inserta en la extrema izquierda de las dos mitades (el nodo RS). (d) El resultado de insertar L en el árbol anterior. La raíz se parte de inmediato, ya que está llena, y el árbol B crece en altura en uno. Luego se inserta L en la hoja que contiene JK. (e) El resultado de insertar F en el árbol anterior. El nodo ABCDE se divide antes de que F se inserte en la más a la derecha de las dos mitades (el nodo DE).

## 18.2-6

Suponga que implementáramos B-ÁRBOL-BÚSQUEDA para utilizar la búsqueda binaria en lugar de la búsqueda lineal dentro de cada nodo. Demuestre que este cambio hace que el tiempo de CPU requerido sea  $O.\lg n/$ , independientemente de cómo pueda elegirse  $t$  como una función de  $n$ .

## 18.2-7

Supongamos que el hardware del disco nos permite elegir el tamaño de una página de disco arbitrariamente, pero que el tiempo que lleva leer la página del disco es  $aCbt$ , donde  $a$  y  $b$  son constantes especificadas y  $t$  es el grado mínimo para un árbol B que usa páginas del tamaño seleccionado. Describa cómo elegir  $t$  para minimizar (aproximadamente) el tiempo de búsqueda del árbol B. Sugiera un valor óptimo de  $t$  para el caso en que  $a = D$  5 milisegundos y  $b = D$  10 microsegundos.

## 18.3 Eliminación de una clave de un árbol B

La eliminación de un árbol B es análoga a la inserción, pero un poco más complicada, porque podemos eliminar una clave de cualquier nodo, no solo una hoja, y cuando eliminamos una clave de un nodo interno, tendremos que reorganizar el nodo. niños. Al igual que en la inserción, debemos cuidarnos de que la eliminación produzca un árbol cuya estructura viole las propiedades del árbol B. Así como teníamos que asegurarnos de que un nodo no se hiciera demasiado grande debido a la inserción, debemos asegurarnos de que un nodo no se hiciera demasiado pequeño durante la eliminación (excepto que se permite que la raíz tenga menos del número mínimo  $t - 1$  de llaves).

Del mismo modo que un algoritmo de inserción simple podría tener que realizar una copia de seguridad si un nodo en la ruta donde se insertaría la clave estaba lleno, un enfoque simple para la eliminación podría tener que realizar una copia de seguridad si un nodo (que no sea la raíz) a lo largo de la ruta a donde se va a borrar la clave tiene el número mínimo de claves.

El procedimiento B-TREE-DELETE borra la clave  $k$  del subárbol enraizado en  $x$ .

Diseñamos este procedimiento para garantizar que siempre que se llame a sí mismo recursivamente en un nodo  $x$ , el número de claves en  $x$  sea al menos el grado mínimo  $t$ . Tenga en cuenta que esta condición requiere una clave más que el mínimo requerido por las condiciones habituales del árbol B, por lo que a veces puede ser necesario mover una clave a un nodo secundario antes de que la recursividad descienda a ese elemento secundario. Esta condición fortalecida nos permite eliminar una clave del árbol en un solo paso hacia abajo sin tener que "retroceder" (con una excepción, que explicaremos). Debe interpretar la siguiente especificación para la eliminación de un árbol B con el entendimiento de que si el nodo raíz  $x$  alguna vez se convierte en un nodo interno sin claves (esta situación puede ocurrir en los casos 2c y 3b en las páginas 501–502), entonces eliminamos  $x$ , y el único hijo de  $x$   $x:c_1$  se convierte en la nueva raíz del árbol, disminuyendo la altura del árbol en uno y conservando la propiedad de que la raíz del árbol contiene al menos una clave (a menos que el árbol esté vacío).

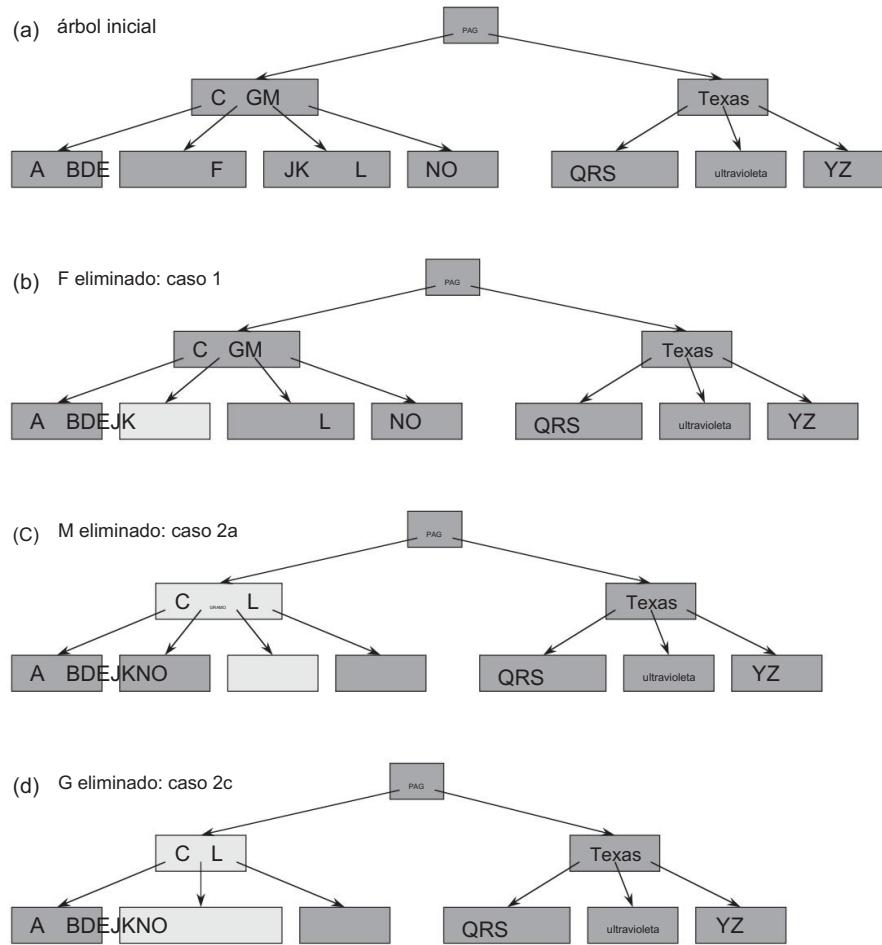


Figura 18.8 Eliminación de claves de un árbol B. El grado mínimo para este árbol B es  $t = 3$ , por lo que un nodo (que no sea la raíz) no puede tener menos de 2 claves. Los nodos que se modifican están ligeramente sombreados. (a) El árbol B de la figura 18.7(e). (b) Supresión de F. Este es el caso 1: supresión simple de una hoja. (c) Eliminación de M. Este es el caso 2a: el predecesor L de M sube para tomar la posición de M. (d) Eliminación de G. Este es el caso 2c: presionamos G hacia abajo para hacer el nodo DEGJK y luego eliminamos G de esta hoja (caso 1).

Esbozamos cómo funciona la eliminación en lugar de presentar el pseudocódigo. La figura 18.8 ilustra los diversos casos de eliminación de claves de un árbol B.

1. Si la clave  $k$  está en el nodo  $x$  y  $x$  es una hoja, elimine la clave  $k$  de  $x$ .
2. Si la clave  $k$  está en el nodo  $x$  y  $x$  es un nodo interno, haga lo siguiente:

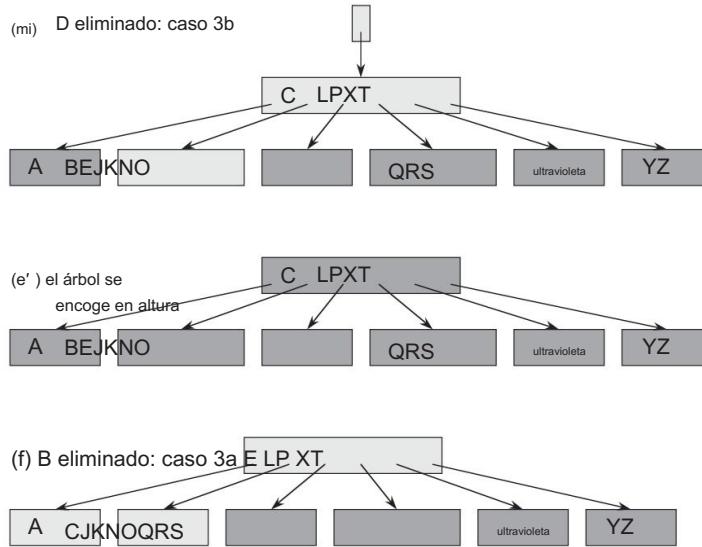


Figura 18.8, continuación (e) Eliminación de D. Este es el caso 3b: la recursividad no puede descender al nodo CL porque tiene solo 2 claves, por lo que presionamos P hacia abajo y lo fusionamos con CL y TX para formar CLP TX; luego eliminamos D de una hoja (caso 1). (e0 ) Después de (e), eliminamos la raíz y el árbol se reduce en uno en altura. (f) Eliminación de B. Este es el caso 3a: C se mueve para ocupar la posición de B y E se mueve para ocupar la posición de C.

- Si el hijo y que precede a k en el nodo x tiene al menos t claves, entonces encuentre predecesor  $k_0$  de k en el subárbol con raíz en y. Eliminar recursivamente  $k_0$ , el y reemplazar k por  $k_0$  en x. (Podemos encontrar  $k_0$  y eliminarlo en un solo paso hacia abajo).
- Si y tiene menos claves que t, entonces, simétricamente, examine el hijo ' que sigue a k en el nodo x. Si ' tiene al menos t claves, busque el sucesor  $k_0$  de k en el subárbol con raíz en '. Elimine recursivamente  $k_0$  y reemplace k por  $k_0$  en x. (Podemos encontrar  $k_0$  y borrarlo en un solo paso hacia abajo.) c. De lo contrario, si tanto y como ' tienen solo t 1 claves, fusione k y todo ' en y, de modo que x pierda tanto k y el puntero a ', y ahora contiene 2t 1 claves. Luego libere ' y borre recursivamente k de y.
- Si la clave k no está presente en el nodo interno x, determine la raíz x:ci del subárbol apropiado que debe contener k, si es que k está en el árbol. Si x:ci tiene solo t 1 claves, ejecute el paso 3a o 3b según sea necesario para garantizar que descendemos a un nodo que contiene al menos t claves. Luego termine recursando al hijo apropiado de x.

- a. Si  $x:ci$  solo tiene  $t - 1$  pero tiene un hermano inmediato con al menos  $t$  llaves, dé a  $x:ci$  una llave extra moviendo una llave de  $x$  hacia abajo a  $x:ci$ , moviendo una llave del hermano inmediato izquierdo o derecho de  $x:ci$  hacia arriba a  $x$ , y moviendo el puntero secundario adecuado del hermano a  $x:ci$ . b. Si  $x:ci$  y los dos hermanos inmediatos de  $x:ci$  tienen  $t - 1$  claves, combine  $x:ci$  con un hermano, lo que implica mover una clave de  $x$  hacia abajo al nuevo nodo fusionado para convertirse en la clave mediana para ese nodo.

Dado que la mayoría de las claves en un árbol B están en las hojas, podemos esperar que en la práctica, las operaciones de eliminación se utilicen con mayor frecuencia para eliminar claves de las hojas. El procedimiento B-TREE-DELETE actúa entonces en un paso hacia abajo a través del árbol, sin tener que retroceder. Sin embargo, al eliminar una clave en un nodo interno, el procedimiento realiza un recorrido descendente por el árbol, pero es posible que tenga que volver al nodo del que se eliminó la clave para reemplazar la clave con su predecesora o sucesora (casos 2a y 2b).

Aunque este procedimiento parece complicado, implica solo operaciones de disco Oh/ para un árbol B de altura h, ya que solo se realizan llamadas O.1/a DISK-READ y DISK WRITE entre invocaciones recursivas del procedimiento. El tiempo de CPU requerido es O.th/ D Ot logt n/.

### Ejercicios

#### 18.3-1

Muestre los resultados de eliminar C, P y V , en orden, del árbol de la figura 18.8(f).

#### 18.3-2

Escriba el pseudocódigo para B-TREE-DELETE.

### Problemas

#### 18-1 Pilas en almacenamiento secundario

Considere implementar una pila en una computadora que tenga una cantidad relativamente pequeña de memoria primaria rápida y una cantidad relativamente grande de almacenamiento en disco más lento. Las operaciones PUSH y POP funcionan en valores de una sola palabra. La pila que deseamos admitir puede crecer mucho más de lo que cabe en la memoria y, por lo tanto, la mayor parte debe almacenarse en el disco.

Una implementación de pila simple, pero ineficiente, mantiene toda la pila en el disco. Mantenemos en la memoria un puntero de pila, que es la dirección de disco del elemento superior de la pila. Si el puntero tiene el valor p, el elemento superior es la palabra .p mod m/th en la página bp=mc del disco, donde m es el número de palabras por página.

Para implementar la operación PUSH , incrementamos el puntero de la pila, leemos la página adecuada en la memoria desde el disco, copiamos el elemento que se va a insertar en la palabra adecuada de la página y escribimos la página de nuevo en el disco. Una operación POP es similar. Disminuimos el puntero de la pila, leemos en la página apropiada del disco y devolvemos la parte superior de la pila. No necesitamos volver a escribir la página, ya que no fue modificada.

Debido a que las operaciones de disco son relativamente costosas, contamos dos costos para cualquier implementación: el número total de accesos al disco y el tiempo total de CPU. Cualquier acceso de disco a una página de m palabras incurre en cargos de un acceso de disco y  $,m/$  tiempo de CPU.

- a. Asintóticamente, ¿cuál es el número de accesos al disco en el peor de los casos para n operaciones de pila usando esta implementación simple? ¿Cuál es el tiempo de CPU para n operaciones de pila? (Exprese su respuesta en términos de m y n para esta parte y las subsiguientes).

Ahora considere una implementación de pila en la que mantenemos una página de la pila en la memoria. (También mantenemos una pequeña cantidad de memoria para realizar un seguimiento de qué página está actualmente en la memoria). Podemos realizar una operación de pila solo si la página de disco relevante reside en la memoria. Si es necesario, podemos escribir la página que está actualmente en la memoria en el disco y leer la nueva página del disco en la memoria. Si la página de disco relevante ya está en la memoria, no se requieren accesos al disco. b. ¿Cuál es el peor número de accesos al disco

necesarios para n PUSH opera ?  
ciones? ¿Cuál es el tiempo de la CPU?

- C. ¿Cuál es el peor número de accesos al disco necesarios para n operaciones de pila?  
¿Cuál es el tiempo de la CPU?

Supongamos que ahora implementamos la pila manteniendo dos páginas en la memoria (además de una pequeña cantidad de palabras para la contabilidad).

- d. Describa cómo administrar las páginas de la pila para que el número amortizado de accesos al disco para cualquier operación de pila sea  $O.1=m/$  y el tiempo de CPU amortizado para cualquier operación de pila sea  $O.1/$ .

#### 18-2 Unión y división de 2-3-4 árboles La operación

de unión toma dos conjuntos dinámicos S0 y S00 y un elemento x tal que para cualquier  $x \in S0$  y  $x \notin S00$ , tenemos  $x :key < x:key < x00: llave$ . Devuelve un conjunto SD  $S0 \cup S00$ . La operación de división es como una unión "inversa": dado un conjunto dinámico S y un elemento x  $\in S$ , crea un conjunto S0 que consta de todos los elementos en S  $\cup S'$  cuyas claves son menores que x:clave y un conjunto S00 que consta de todos los elementos en S  $\cup S'$  cuyas claves son mayores que x:clave. En este problema investigamos

cómo implementar estas operaciones en 2-3-4 árboles. Asumimos por conveniencia que los elementos consisten solo en claves y que todos los valores clave son distintos.

- a. Muestre cómo mantener, para cada nodo  $x$  de un árbol 2-3-4, la altura del subárbol con raíz en  $x$  como un atributo  $x:\text{altura}$ . Asegúrese de que su implementación no afecte los tiempos de ejecución asintóticos de búsqueda, inserción y eliminación.
- b. Muestre cómo implementar la operación de unión. Dados dos árboles 2-3-4  $T$  y una clave  $k$ , la operación de unión debería ejecutarse en el tiempo  $O(1)$   $C_{jh0} h0j/$ , donde  $h0$  y  $h00$  son las alturas de  $T$  y  $T$   $00$ , respectivamente.
- C. Considere el camino simple  $p$  desde la raíz de un árbol 2-3-4  $T$  hasta una clave dada  $k$ , el conjunto  $S0$  de claves en  $T$  que son menores que  $k$ , y el conjunto  $S00$  de claves en  $T$  que son mayores que  $k$ . Muestre que  $p$  descompone  $S0$  en un conjunto de árboles  $T_1^0; \dots; T_m^0$  y un juego de llaves  $f(k_0; k_0 2; \dots; k_0 m)$ , donde, para  $i \in \{1, 2, \dots, m\}$ , tenemos  $y < k_0$  y  $y > T_i$ . ¿Cuál es la relación entre las alturas? Describa cómo  $p$  divide  $S00$  de  $T_1^0$  y  $T_m^0$  en conjuntos de árboles y claves.
- d. Muestre cómo implementar la operación de división en  $T$ . Utilice la operación de combinación para ensamblar las claves en  $S0$  en un solo árbol 2-3-4  $T$  y las claves en  $S00$  en un solo árbol 2-3-4  $T$   $00$ . El tiempo de ejecución de la operación dividida debe ser  $O(\lg n)$ , donde  $n$  es el número de claves en  $T$ . (Sugerencia: los costos para unirse deben ser telescopicos).

## Notas del capítulo

Knuth [211], Aho, Hopcroft y Ullman [5], y Sedgewick [306] brindan más discusiones sobre esquemas de árboles balanceados y árboles B. Comer [74] ofrece un estudio exhaustivo de los árboles B. Guibas y Sedgewick [155] analizan las relaciones entre varios tipos de esquemas de árboles equilibrados, incluidos los árboles rojo-negro y 2-3-4.

En 1970, JE Hopcroft inventó los árboles 2-3, un precursor de los árboles B y los árboles 2-3-4, en los que cada nodo interno tiene dos o tres hijos. Bayer y McCreight [35] introdujeron los árboles B en 1972; no explicaron su elección de nombre.

Bender, Demaine y Farach-Colton [40] estudiaron cómo hacer que los árboles B funcionen bien en presencia de efectos de jerarquía de memoria. Sus algoritmos que olvidan la memoria caché funcionan de manera eficiente sin conocer explícitamente los tamaños de transferencia de datos dentro de la jerarquía de la memoria.

---

## 19

## Montones de Fibonacci

La estructura de datos del montón de Fibonacci tiene un doble propósito. Primero, admite un conjunto de operaciones que constituye lo que se conoce como un "montón combinable". En segundo lugar, varias operaciones del montón de Fibonacci se ejecutan en un tiempo amortizado constante, lo que hace que esta estructura de datos sea adecuada para aplicaciones que invocan estas operaciones con frecuencia.

### Montones combinables

Un montón fusionable es cualquier estructura de datos que admita las siguientes cinco operaciones, en las que cada elemento tiene una clave:

MAKE-HEAP./ crea y devuelve un nuevo montón que no contiene elementos.

INSERTAR.H; x/ inserta el elemento x, cuya clave ya se ha rellenado, en el montón H.

MINIMUM.H / devuelve un puntero al elemento en el montón H cuya clave es mínima.

EXTRACT-MIN.H / elimina el elemento del montón H cuya clave es mínima, re  
girando un puntero al elemento.

UNIÓN.H1; H2/ crea y devuelve un nuevo montón que contiene todos los elementos de los montones H1  
y H2. Esta operación "destruye" los montones H1 y H2 .

Además de las operaciones de almacenamiento dinámico combinables anteriores, los almacenamientos dinámicos de Fibonacci  
también admiten las siguientes dos operaciones:

DISMINUIR-TECLA.H; X; k/ asigna al elemento x dentro del montón H el nuevo valor de clave k, que  
suponemos que no es mayor que su valor de clave actual.<sup>1</sup>

ELIMINAR.H; x/ elimina el elemento x del montón H.

---

<sup>1</sup>Como se mencionó en la introducción a la Parte V, nuestros montones combinables predeterminados son  
montones mínimos combinables, por lo que se aplican las operaciones MINIMUM, EXTRACT-MIN y DECREASE-  
KEY . Alternativamente, podríamos definir un montón máximo fusionable con las operaciones MAXIMUM,  
EXTRACT-MAX e INCREASE-KEY .

Procedimiento	Montón binario Montón de Fibonacci (en el peor de los casos) (amortizado)
HACER-MONTÓN	, $\lg n$ / ,1/ ,1/ ,1/ , $\lg n$ / O. $\lg n$
INSERTAR	, $n$ / ,1/ , $\lg n$ / ,1/ , $\lg n$ / O. $\lg n$
MÍNIMO	
EXTRACTO-MIN	
UNIÓN	
DISMINUIR-TECLA	
BORRAR	

Figura 19.1 Tiempos de ejecución para operaciones en dos implementaciones de montones fusionables. El número de elementos en el(es) montón(es) en el momento de una operación se denota por  $n$ .

Como muestra la tabla de la figura 19.1, si no necesitamos la operación UNION, los montones binarios ordinarios, como los que se usan en heapsort (capítulo 6), funcionan bastante bien. Las operaciones que no sean UNION se ejecutan en el peor de los casos  $O.\lg n$  en un montón binario. Sin embargo, si necesitamos admitir la operación UNION, los montones binarios funcionan mal. Al concatenar las dos matrices que contienen los montones binarios que se fusionarán y luego ejecutar BUILD-MIN-HEAP (consulte la Sección 6.3), la operación UNION toma , $n$ / tiempo en el peor de los casos.

Los montones de Fibonacci, por otro lado, tienen mejores límites de tiempo asintóticos que los montones binarios para las operaciones INSERT, UNION y DECREASE-KEY, y tienen los mismos tiempos de ejecución asintóticos para las operaciones restantes. Tenga en cuenta, sin embargo, que los tiempos de ejecución de los montones de Fibonacci en la figura 19.1 son límites de tiempo amortizados, no límites de tiempo por operación en el peor de los casos. La operación UNION solo toma un tiempo constante amortizado en un montón de Fibonacci, que es significativamente mejor que el tiempo lineal en el peor de los casos requerido en un montón binario (asumiendo, por supuesto, que un límite de tiempo amortizado es suficiente).

### Montones de Fibonacci en teoría y práctica

Desde un punto de vista teórico, los montones de Fibonacci son especialmente deseables cuando el número de operaciones EXTRACT-MIN y DELETE es pequeño en relación con el número de otras operaciones realizadas. Esta situación se presenta en muchas aplicaciones. Por ejemplo, algunos algoritmos para problemas de gráficos pueden llamar a DECREASE-KEY una vez por borde. Para gráficos densos, que tienen muchos bordes, el ,1/ tiempo amortizado de cada llamada de DECREASE-KEY suma una gran mejora con respecto al , $\lg n$  / tiempo en el peor de los casos de montones binarios. Los algoritmos rápidos para problemas como el cálculo de árboles de expansión mínimos (Capítulo 23) y la búsqueda de rutas más cortas de fuente única (Capítulo 24) hacen un uso esencial de los montones de Fibonacci.

Sin embargo, desde un punto de vista práctico, los factores constantes y la complejidad de programación de los montones de Fibonacci los hacen menos deseables que los montones binarios (o k-arios) ordinarios para la mayoría de las aplicaciones, excepto para ciertas aplicaciones que manejan grandes cantidades de datos. Por lo tanto, los montones de Fibonacci son predominantemente de interés teórico. Si se desarrollara una estructura de datos mucho más simple con los mismos límites de tiempo amortizados que los montones de Fibonacci, también sería de utilidad práctica.

Tanto los montones binarios como los montones de Fibonacci son ineficientes en la forma en que admiten la operación BUSCAR; puede llevar un tiempo encontrar un elemento con una clave determinada. Por esta razón, operaciones como DECREASE-KEY y DELETE que se refieren a un elemento dado requieren un puntero a ese elemento como parte de su entrada. Como en nuestra discusión de las colas de prioridad en la Sección 6.5, cuando usamos un montón combinable en una aplicación, a menudo almacenamos un identificador para el objeto de la aplicación correspondiente en cada elemento del montón combinable, así como un identificador para el elemento del montón combinable correspondiente en cada objeto de aplicación. La naturaleza exacta de estos identificadores depende de la aplicación y su implementación.

Como muchas otras estructuras de datos que hemos visto, los montones de Fibonacci se basan en árboles enraizados. Representamos cada elemento por un nodo dentro de un árbol, y cada nodo tiene un atributo clave. En el resto de este capítulo, utilizaremos el término "nodo" en lugar de "elemento". También ignoraremos los problemas de asignación de nodos antes de la inserción y liberación de nodos después de la eliminación, suponiendo en cambio que el código que llama a los procedimientos del montón se ocupa de estos detalles.

La sección 19.1 define los montones de Fibonacci, analiza cómo los representamos y presenta la función potencial utilizada para su análisis amortizado. La sección 19.2 muestra cómo implementar las operaciones de almacenamiento dinámico fusionable y lograr los límites de tiempo amortizados que se muestran en la figura 19.1. Las dos operaciones restantes, DECREASE KEY y DELETE, forman el enfoque de la Sección 19.3. Finalmente, la Sección 19.4 finaliza una parte clave del análisis y también explica el curioso nombre de la estructura de datos.

## 19.1 Estructura de los montones de Fibonacci

Un montón de Fibonacci es una colección de árboles enraizados que están ordenados por montones mínimos. Es decir, cada árbol obedece a la propiedad min-heap: la clave de un nodo es mayor o igual que la clave de su padre. La figura 19.2(a) muestra un ejemplo de un montón de Fibonacci.

Como muestra la figura 19.2(b), cada nodo  $x$  contiene un puntero  $x:p$  a su padre y un puntero  $x:hijo$  a cualquiera de sus hijos. Los hijos de  $x$  están enlazados en una lista circular doblemente enlazada, a la que llamamos lista de hijos de  $x$ . Cada hijo  $y$  en una lista de hijos tiene punteros  $y:left$  e  $y:right$  que apuntan a los hermanos izquierdo y derecho de  $y$ , respectivamente. Si el nodo  $y$  es hijo único, entonces  $y:left = y:right = y$ . Los hermanos pueden aparecer en una lista de niños en cualquier orden.

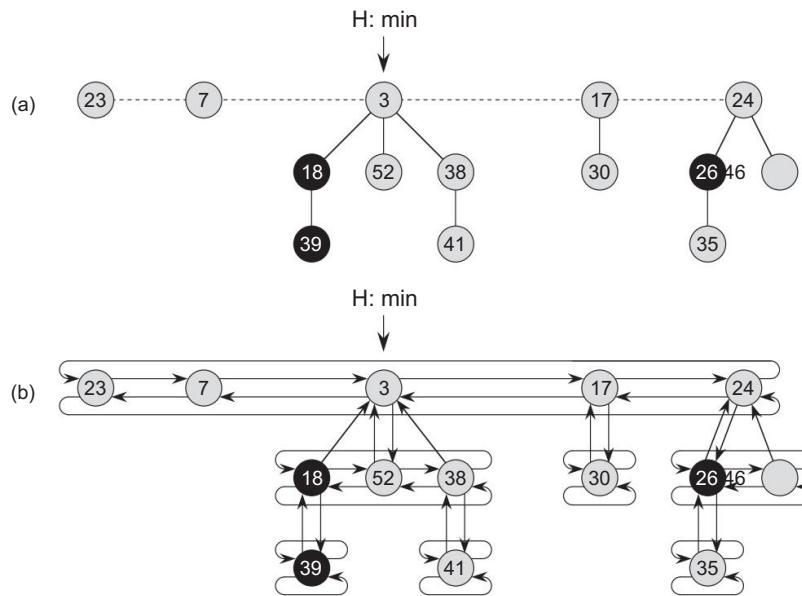


Figura 19.2 (a) Un montón de Fibonacci que consta de cinco árboles ordenados por montón mínimo y 14 nodos. La línea discontinua indica la lista raíz. El nodo mínimo del montón es el nodo que contiene la clave 3.

Los nodos negros están marcados. El potencial de este montón de Fibonacci en particular es  $5C2 3 D 11$ . (b) Una representación más completa que muestra los punteros  $p$  (flechas hacia arriba), hijo (flechas hacia abajo) e izquierda y derecha (flechas hacia los lados). Las figuras restantes de este capítulo omiten estos detalles, ya que toda la información aquí mostrada se puede determinar a partir de lo que aparece en el inciso (a).

Las listas circulares doblemente enlazadas (consulte la Sección 10.2) tienen dos ventajas para su uso en los montones de Fibonacci. Primero, podemos insertar un nodo en cualquier ubicación o eliminar un nodo de cualquier lugar en una lista circular doblemente enlazada en tiempo  $O(1)$ . En segundo lugar, dadas dos listas de este tipo, podemos concatenarlas (o “empalmarlas”) en una lista circular doblemente enlazada en tiempo  $O(1)$ . En las descripciones de las operaciones de montón de Fibonacci, nos referiremos a estas operaciones de manera informal, lo que le permitirá completar los detalles de sus implementaciones si lo desea.

Cada nodo tiene otros dos atributos. Almacenamos el número de hijos en la lista de hijos del nodo  $x$  en  $x:\text{grado}$ . El atributo de valor booleano  $x:\text{mark}$  indica si el nodo  $x$  ha perdido un hijo desde la última vez que  $x$  se convirtió en hijo de otro nodo.

Los nodos recién creados no están marcados, y un nodo  $x$  deja de estar marcado cada vez que se convierte en hijo de otro nodo. Hasta que analicemos la operación DECREASE-KEY en la Sección 19.3, estableceremos todos los atributos de marca en FALSO.

Accedemos a un montón de Fibonacci  $H$  mediante un puntero  $H:\text{min}$  a la raíz de un árbol que contiene la clave mínima; llamamos a este nodo el nodo mínimo de Fibonacci

montón. Si más de una raíz tiene una clave con el valor mínimo, entonces cualquiera de esas raíces puede servir como nodo mínimo. Cuando un montón de Fibonacci H está vacío, H:min es NIL.

Las raíces de todos los árboles en un montón de Fibonacci se vinculan mediante sus punteros izquierdo y derecho en una lista circular doblemente vinculada llamada lista raíz del montón de Fibonacci. El puntero H:min apunta al nodo en la lista raíz cuya clave es mínima. Los árboles pueden aparecer en cualquier orden dentro de una lista raíz.

Confiamos en otro atributo para un montón de Fibonacci H: H:n, la cantidad de nodos actualmente en H.

#### Función potencial

Como se mencionó, usaremos el método potencial de la Sección 17.3 para analizar el desempeño de las operaciones del montón de Fibonacci. Para un montón de Fibonacci H, indicamos por tH / el número de árboles en la lista raíz de H y por mH / el número de nodos marcados en H. Luego definimos el potencial  $\hat{H}$  / del montón de Fibonacci H por

$$\hat{H} / D tH / C 2 mH : \quad (19.1)$$

(Obtendremos algo de intuición para esta función potencial en la Sección 19.3.) Por ejemplo, el potencial del montón de Fibonacci que se muestra en la Figura 19.2 es 5C 2 3 D 11. El potencial de un conjunto de montones de Fibonacci es la suma de los potenciales de sus montones constituyentes de Fibonacci. Supondremos que una unidad de potencial puede pagar una cantidad constante de trabajo, donde la constante es lo suficientemente grande como para cubrir el costo de cualquiera de los trabajos específicos de tiempo constante que podamos encontrar.

Suponemos que una aplicación de montón de Fibonacci comienza sin montones. Por lo tanto, el potencial inicial es 0 y, por la ecuación (19.1), el potencial es no negativo en todos los tiempos subsiguientes. A partir de la ecuación (17.3), un límite superior del costo total amortizado proporciona un límite superior del costo real total para la secuencia de operaciones.

#### Grado máximo

Los análisis amortizados que realizaremos en las secciones restantes de este capítulo suponen que conocemos un límite superior  $D_n$  / en el grado máximo de cualquier nodo en un montón de Fibonacci de n nodos. No lo probaremos, pero cuando solo se admiten las operaciones de almacenamiento dinámico fusionables,  $D_n / \lg n$ . (El problema 19-2(d) le pide que demuestre esta propiedad.) En las secciones 19.3 y 19.4, mostraremos que cuando admitimos DECREASE-KEY y DELETE también,  $D_n / O.\lg n$ .

## 19.2 Operaciones de almacenamiento dinámico fusionable

Las operaciones de almacenamiento dinámico fusionable en los almacenamientos dinámicos de Fibonacci retrasan el trabajo tanto como sea posible.

Las diversas operaciones tienen compensaciones de rendimiento. Por ejemplo, insertamos un nodo agregándolo a la lista raíz, lo que requiere un tiempo constante. Si empezáramos con un montón de Fibonacci vacío y luego insertáramos k nodos, el montón de Fibonacci consistiría solo en una lista raíz de k nodos. La compensación es que si luego realizamos una operación EXTRACT-MIN en el montón H de Fibonacci, después de eliminar el nodo al que apunta H:min , tendríamos que mirar a través de cada uno de los k 1 nodos restantes en la lista raíz para encontrar el nuevo nodo mínimo. Siempre que tengamos que pasar por toda la lista de raíces durante la operación EXTRACT-MIN , también consolidaremos los nodos en árboles ordenados por montones mínimos para reducir el tamaño de la lista de raíces. Veremos que, independientemente del aspecto de la lista raíz antes de una operación EXTRACT-MIN , después cada nodo de la lista raíz tiene un grado que es único dentro de la lista raíz, lo que conduce a una lista raíz de tamaño como máximo Dn/ C 1.

## Crear un nuevo montón de Fibonacci

Para crear un montón de Fibonacci vacío, el procedimiento MAKE-FIB-HEAP asigna y devuelve el objeto H del montón de Fibonacci, donde H:n D 0 y H:min D NIL; no hay árboles en H. Debido a que tH / D 0 y mH / D 0, el potencial del montón vacío de Fibonacci es  $\cdot H / D 0$ . El costo amortizado de MAKE-FIB-HEAP es, por lo tanto, igual a su O.1 / costo real.

## Insertar un nodo

El siguiente procedimiento inserta el nodo x en el montón H de Fibonacci, asumiendo que el nodo ya se ha asignado y que x:key ya se ha completado.

FIB-MONTAJE-INSERTAR.H; x/

1 x:grado D 0 2 x:p D

NIL 3 x:niño D NIL

4 x:marca D FALSO 5

si H:min == NIL 6 7

crear una lista raíz para H que contenga solo x  
H:min D x 8

sino inserte x en la lista raíz de H

si x:clave < H:min:clave

9 10                   Alt.:mín. Prof. x

11 H:n DH:n C 1

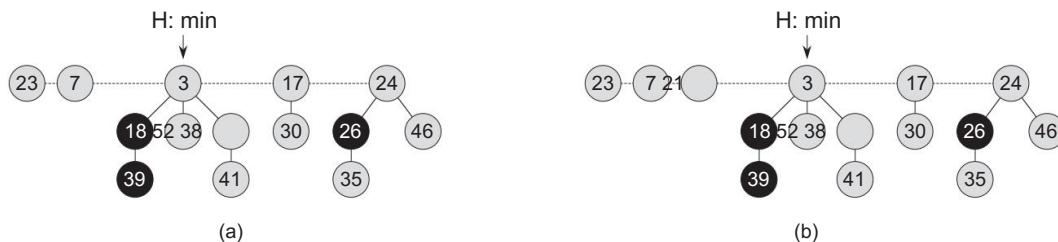


Figura 19.3 Inserción de un nodo en un montón de Fibonacci. (a) Un montón H de Fibonacci. (b) Un montón H de Fibonacci después de insertar el nodo con la clave 21. El nodo se convierte en su propio árbol ordenado por montón mínimo y luego se agrega a la lista raíz, convirtiéndose en el hermano izquierdo de la raíz.

Las líneas 1 a 4 inicializan algunos de los atributos estructurales del nodo x. Línea 5 pruebas para ver si el montón H de Fibonacci está vacío. Si es así, las líneas 6 y 7 hacen que x sea el único nodo en la lista raíz de H y establecen H:min para que apunte a x. De lo contrario, las líneas 8 a 10 insertan x en la lista raíz de H y actualizan H:min si es necesario. Finalmente, la línea 11 incrementa H:n para reflejar la adición del nuevo nodo. La figura 19.3 muestra un nodo con la clave 21 insertada en el montón de Fibonacci de la figura 19.2.

Para determinar el costo amortizado de FIB-HEAP-INSERT, sea  $H$  el montón de Fi bonacci de entrada y  $H_0$  el montón de Fibonacci resultante. Entonces,  $t.H_0 / D \leq t.H / C + m.H_0 / D + mH / C$ , y el aumento de potencial es

..tH / C 1/ C 2 mH // .tH / C 2 mH // D 1

Dado que el costo real es Q.1/, el costo amortizado es Q.1/C.1 D.Q.1/

### Encontrar el nodo mínimo

El nodo mínimo de un montón de Fibonacci H viene dado por el puntero  $H:\min$ , por lo que podemos encontrar el nodo mínimo en  $O.1/\sqrt{n}$  tiempo real. Como el potencial de  $H$  no cambia, el costo amortizado de esta operación es igual a su  $O.1/\sqrt{n}$  costo real.

### Uniendo dos montones de Fibonacci

El siguiente procedimiento une los montones de Fibonacci H1 y H2, destruyendo H1 y H2 en el proceso. Simplemente concatena las listas raíz de H1 y H2 y luego determina el nuevo nodo mínimo. Posteriormente, los objetos que representan H1 y H2 nunca se volverán a utilizar.

```

FIB-MONTÓN-UNIÓN.H1; H2/
1 HD MAKE-FIB-HEAP./ 2 H:min
D H1:min 3 concatenar
la lista raíz de H2 con la lista raíz de H 4 si .H1:min == NIL/
o .H2:min ≠ NIL y H2:min:tecla < H1:min:tecla/ 5 H:min D H2:min 6 H:n D H1:n
C H2:n 7 volver H

```

Las líneas 1 a 3 concatenan las listas raíz de H1 y H2 en una nueva lista raíz H. Las líneas 2, 4 y 5 establecen el nodo mínimo de H y la línea 6 establece H:n en el número total de nodos. La línea 7 devuelve el montón H de Fibonacci resultante. Al igual que en el procedimiento FIB-HEAP INSERT , todas las raíces siguen siendo raíces.

El cambio de potencial es

```

^.H / ^.H1/ C ^.H2// D .tH / C 2
mH // ..t.H1/ C 2 m.H1// C .t.H2/ C 2 m.H2// / D 0
:

```

porque tH / D t.H1/ C t.H2/ y mH / D m.H1/ C m.H2/. El coste amortizado de FIB-HEAP-UNION es por tanto igual a su coste real O,1/.

Extrayendo el nodo mínimo

El proceso de extracción del nodo mínimo es la más complicada de las operaciones presentadas en esta sección. También es donde finalmente ocurre el trabajo retrasado de consolidar árboles en la lista de raíces. El siguiente pseudocódigo extrae el nodo mínimo. El código asume por conveniencia que cuando se elimina un nodo de una lista enlazada, los punteros restantes en la lista se actualizan, pero los punteros en el nodo extraído no se modifican. También llama al procedimiento auxiliar CONSOLIDAR, que veremos en breve.

```

FIB-HEAP-EXTRACT-MIN.H /
1   D H:min 2 si '
  ☐ NIL 3 para
    cada niño x de ' 4 agregue x a
    la lista raíz de H 5 6 7 8
      x:p D NIL
      eliminar ' de la lista raíz de H if ' == =
        ':right H:min D
        NIL else H:min
9     D ':right CONSOLIDATE.H /
10
11     H:n DH:n 1
12 vuelta '

```

Como ilustra la Figura 19.4, FIB-HEAP-EXTRACT-MIN funciona haciendo primero una raíz de cada uno de los hijos del nodo mínimo y eliminando el nodo mínimo de la lista raíz. Luego consolida la lista de raíces enlazando raíces de igual grado hasta que como máximo quede una raíz de cada grado.

Comenzamos en la línea 1 guardando un puntero ' al nodo mínimo; el procedimiento devuelve este puntero al final. Si ' es NIL, entonces el montón H de Fibonacci ya está vacío y hemos terminado. De lo contrario, eliminamos el nodo ' de H haciendo que todos los hijos de ' sean raíces de H en las líneas 3 a 5 (colocándolos en la lista raíz) y eliminando ' de la lista raíz en la línea 6. Si ' es por derecho propio hermano después de la línea 6, entonces ' era el único nodo en la lista raíz y no tenía hijos, por lo que todo lo que queda es hacer que el montón de Fibonacci esté vacío en la línea 8 antes de regresar '. De lo contrario, establecemos el puntero H:min en la lista raíz para que apunte a una raíz que no sea ' (en este caso, el hermano derecho de '), que no necesariamente será el nuevo nodo mínimo cuando FIB-HEAP-EXTRACT -MIN está hecho. La figura 19.4(b) muestra el montón de Fibonacci de la figura 19.4(a) después de ejecutar la línea 9.

El siguiente paso, en el que reducimos el número de árboles en el montón de Fibonacci, es consolidar la lista de raíces de H, lo que se logra con la llamada CONSOLIDATE.H /. La consolidación de la lista de raíces consiste en ejecutar repetidamente los siguientes pasos hasta que cada raíz en la lista de raíces tenga un valor de grado distinto:

1. Encuentra dos raíces x e y en la lista de raíces con el mismo grado. sin pérdida de generalidad, sea x:key y:key.
2. Vincule y con x: elimine y de la lista raíz y conviértalo en un hijo de x llamando al procedimiento FIB-HEAP-LINK . Este procedimiento incrementa el atributo x:grado y borra la marca en y.

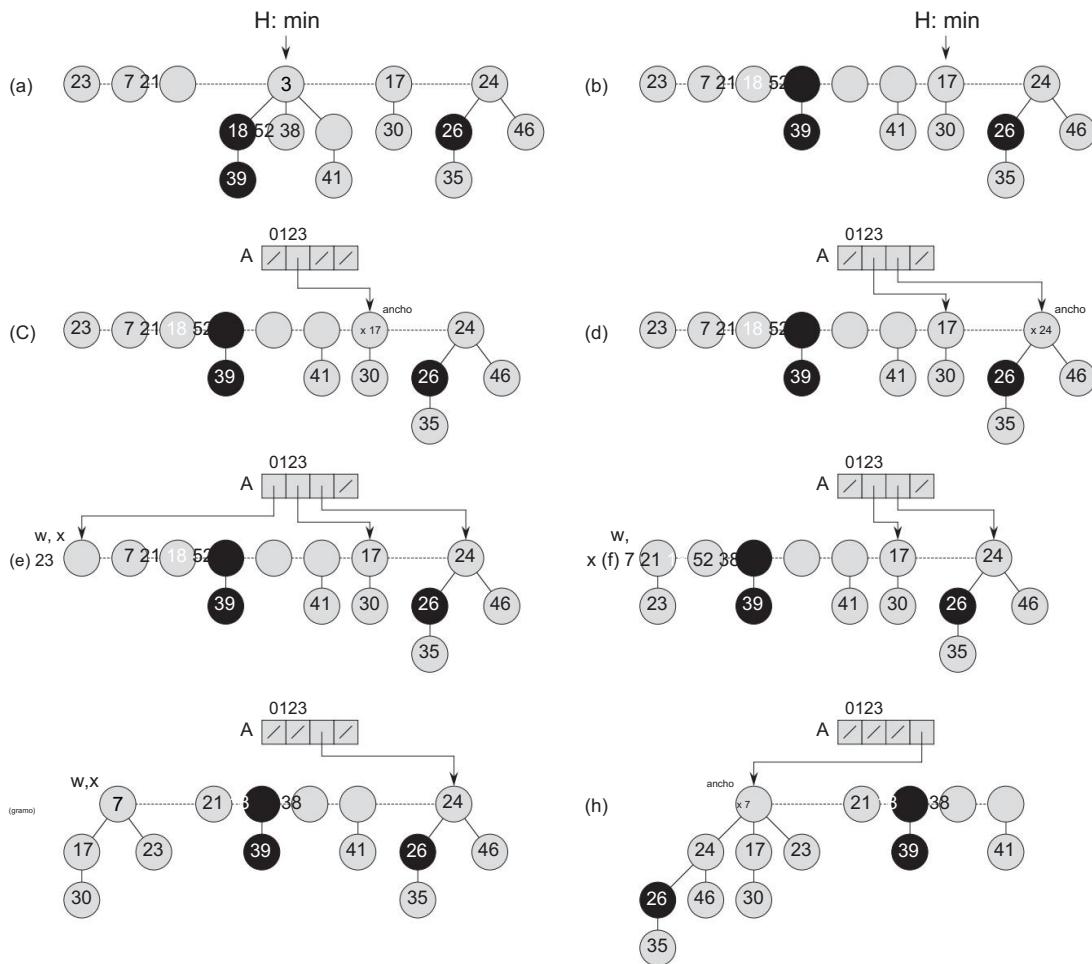


Figura 19.4 La acción de FIB-HEAP-EXTRACT-MIN. (a) Un montón H de Fibonacci. (b) La situación después de eliminar el nodo mínimo de la lista raíz y agregar sus hijos a la lista raíz. (c)–(e) El arreglo A y los árboles después de cada una de las primeras tres iteraciones del ciclo for de las líneas 4–14 del procedimiento CONSOLIDAR. El procedimiento procesa la lista raíz comenzando en el nodo al que apunta H:min y siguiendo los punteros a la derecha. Cada parte muestra los valores de w y x al final de una iteración. (f)–(h) La siguiente iteración del ciclo for , con los valores de w y x mostrados al final de cada iteración del ciclo while de las líneas 7–13. La parte (f) muestra la situación después de la primera vez a través del bucle while . El nodo con la clave 23 se ha vinculado al nodo con la clave 7, al que apunta x ahora.

En la parte (g), el nodo con la clave 17 se ha vinculado al nodo con la clave 7, al que todavía apunta x. En la parte (h), el nodo con la clave 24 se ha vinculado al nodo con la clave 7. Dado que AŒ3 no apuntó previamente a ningún nodo, al final de la iteración del bucle for, AŒ3 se configura para que apunte a la raíz del resultante árbol.

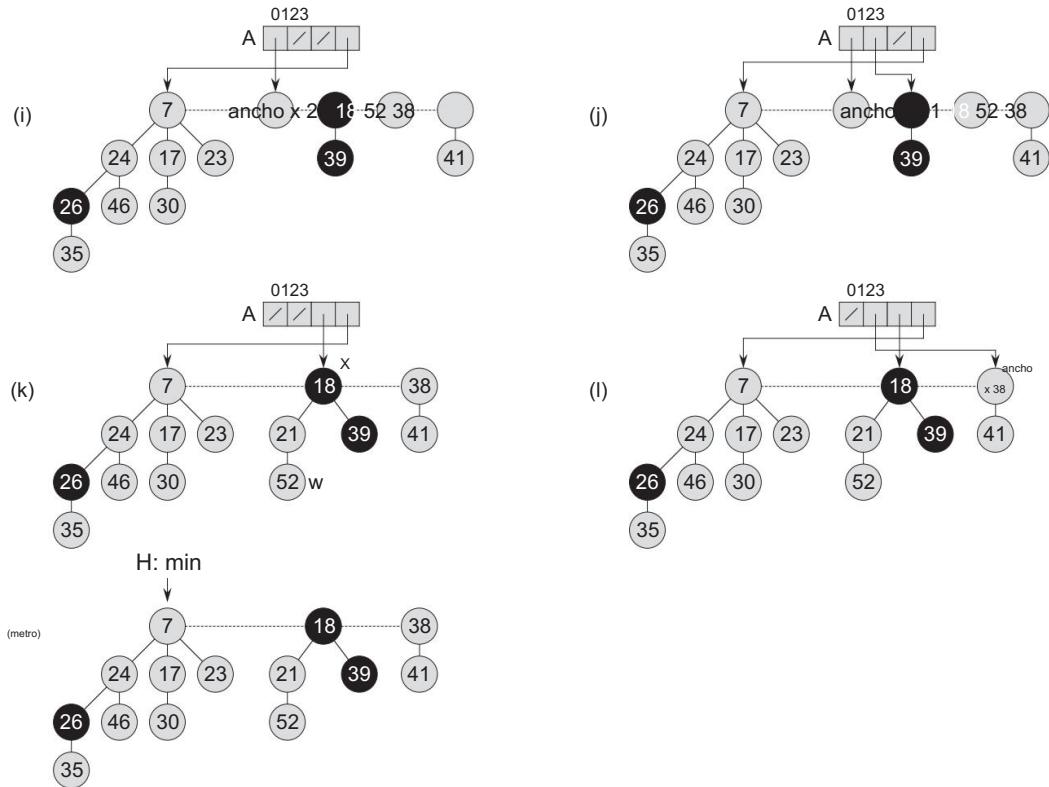


Figura 19.4, continuación (i)–(l) La situación después de cada una de las próximas cuatro iteraciones del bucle for . (m) Montón H de Fibonacci después de reconstruir la lista raíz a partir de la matriz A y determinar el nuevo puntero H:min .

El procedimiento CONSOLIDATE utiliza una matriz auxiliar  $A[0:n] : [DH:n]$  para realizar un seguimiento de las raíces según sus grados. Si  $A[i] \in D$  y, entonces y actualmente es una raíz con  $y: grado D$ . Por supuesto, para asignar el arreglo tenemos que saber cómo calcular el límite superior  $DH:n$  en el grado máximo, pero veremos cómo hacerlo en la Sección 19.4.

CONSOLIDATE.H / 1 sea

```

AŒ0 : : DH:n/ un nuevo arreglo 2 para i D 0 a
DH:n/ 3 AŒi D NIL 4 para cada
nodo w en la lista raíz de
H 5 6 7 8

x fondo ancho

d D x:grado while
AŒd ≠ NIL y D AŒd //
otro nodo con el mismo grado que x si x:clave > y:clave intercambia x con y FIB-
9      HEAP-LINK.H; y; x/
10     AŒd D NIL d D d C 1
11     AŒd D x 14 15 H:min D NIL 16 for
12     i D 0 to DH:n/ if
13     AŒi ≠ NIL 17

if H:min ==

NIL cree una lista raíz

para H que contenga solo AŒi
H:min D AŒi sino
inserte AŒi en la lista raíz de H si
19           AŒi:key < H:min:key H:min D AŒi
20
21
22
23

```

FIB-HEAP-ENLACE.H; y; x/ 1

eliminar y de la lista raíz de H 2 convertirlo en  
hijo de x, incrementando x:grado 3 y:marcar D FALSO

En detalle, el procedimiento CONSOLIDAR funciona de la siguiente manera. Las líneas 1 a 3 asignan e inicializan la matriz A haciendo que cada entrada sea NIL. El ciclo for de las líneas 4 a 14 procesa cada raíz w en la lista de raíces. A medida que vinculamos las raíces, w puede estar vinculado a algún otro nodo y dejar de ser una raíz. Sin embargo, w siempre está en un árbol con raíz en algún nodo x, que puede ser o no el mismo w. Como queremos como máximo una raíz con cada grado, buscamos en el arreglo A para ver si contiene una raíz y con el mismo grado que x. Si es así, vinculamos las raíces x e y, pero garantizando que x siga siendo una raíz después de la vinculación. Es decir, vinculamos y con x después de intercambiar primero los punteros a las dos raíces si la clave de y es más pequeña que la clave de x. Después de vincular y con x, el grado de x ha aumentado en 1, por lo que continuamos este proceso, vinculando x y otra raíz cuyo grado es igual al nuevo grado de x, hasta que ninguna otra raíz

que hemos procesado tiene el mismo grado que  $x$ . Luego establecemos la entrada apropiada de  $A$  para que apunte a  $x$ , de modo que, a medida que procesemos las raíces más adelante, hayamos registrado que  $x$  es la única raíz de su grado que ya hemos procesado. Cuando este bucle for termine, quedará como máximo una raíz de cada grado, y la matriz  $A$  apuntará a cada raíz restante.

El bucle while de las líneas 7 a 13 vincula repetidamente la raíz  $x$  del árbol que contiene el nodo  $w$  con otro árbol cuya raíz tiene el mismo grado que  $x$ , hasta que ninguna otra raíz tiene el mismo grado. Este ciclo while mantiene el siguiente invariante: Al inicio de cada iteración

del ciclo while ,  $d D x:\text{grado}$ .

Usamos este ciclo invariante de la siguiente manera:

Inicialización: la línea 6 asegura que el bucle se mantiene invariante la primera vez que ingresamos el lazo.

Mantenimiento: en cada iteración del ciclo while ,  $A \leftarrow d$  apunta a alguna raíz  $y$ .

Como  $d D x:\text{grado} D y:\text{grado}$ , queremos vincular  $x$  e  $y$ . Cualquiera de  $x$  e  $y$  que tenga la clave más pequeña se convierte en el padre del otro como resultado de la operación de enlace, por lo que las líneas 9 y 10 intercambian los punteros a  $x$  e  $y$  si es necesario.

A continuación, vinculamos  $y$  con  $x$  mediante la llamada FIB-HEAP-LINK. $H; y; x/$  en la línea 11. Esta llamada incrementa  $x:\text{grado}$  pero deja  $y:\text{grado}$  como  $d$ . El nodo  $y$  ya no es una raíz, por lo que la línea 12 elimina el puntero a él en la matriz  $A$ . Debido a que la llamada de FIB-HEAP-LINK incrementa el valor de  $x:\text{grado}$ , la línea 13 restaura el invariante que  $d D x:\text{grado}$ .

Terminación: Repetimos el ciclo while hasta  $A \leftarrow d$  NIL, en cuyo caso hay ninguna otra raíz con el mismo grado que  $x$ .

Después de que termina el ciclo while , establecemos  $A \leftarrow d$  en  $x$  en la línea 14 y realizamos la siguiente iteración del ciclo for .

Las figuras 19.4(c)–(e) muestran el arreglo  $A$  y los árboles resultantes después de las primeras tres iteraciones del ciclo for de las líneas 4–14. En la siguiente iteración del bucle for , se producen tres enlaces; sus resultados se muestran en las Figuras 19.4(f)–(h). Las figuras 19.4(i)–(l) muestran el resultado de las próximas cuatro iteraciones del bucle for .

Todo lo que queda es limpiar. Una vez que se completa el bucle for de las líneas 4 a 14, la línea 15 vacía la lista raíz y las líneas 16 a 23 la reconstruyen a partir del arreglo  $A$ . El montón de Fibonacci resultante aparece en la figura 19.4(m). Después de consolidar la lista raíz, FIB-HEAP-EXTRACT-MIN termina disminuyendo  $H:n$  en la línea 11 y devolviendo un puntero al nodo eliminado ' en la línea 12.

Ahora estamos listos para demostrar que el costo amortizado de extraer el nodo mínimo de un montón de Fibonacci de  $n$  nodos es  $O(Dn)$ . Sea  $H$  el montón de Fibonacci justo antes de la operación FIB-HEAP-EXTRACT-MIN .

Comenzamos por contabilizar el costo real de extraer el nodo mínimo. Una contribución de  $O(Dn)$  proviene del procesamiento FIB-HEAP-EXTRACT-MIN en

la mayoría de los hijos  $D_n$  del nodo mínimo y del trabajo en las líneas 2–3 y 16–23 de CONSOLIDATE. Queda por analizar la contribución del ciclo for de las líneas 4–14 en CONSOLIDATE, para lo cual usamos un análisis agregado. El tamaño de la lista raíz al llamar a CONSOLIDATE es como máximo  $D_n / C tH / 1$ , ya que consiste en los nodos originales  $tH / root-list$ , menos el nodo raíz extraído, más los hijos del nodo extraído, cuyo número como máximo  $D_n$ . Dentro de una iteración dada del bucle for de las líneas 4 a la 14, el número de iteraciones del bucle while de las líneas 7 a la 13 depende de la lista raíz. Pero sabemos que cada vez que se ejecuta el ciclo while, una de las raíces está vinculada a otra y, por lo tanto, el número total de iteraciones del ciclo while sobre todas las iteraciones del ciclo for es, como máximo, el número de raíces en la lista de raíces. Por lo tanto, la cantidad total de trabajo realizado en el bucle for es, como mucho, proporcional a  $D_n / C tH / 1$ . Por lo tanto, el trabajo real total para extraer el nodo mínimo es  $O(D_n / C tH / 1)$ .

El potencial antes de extraer el nodo mínimo es  $tH / C 2 mH /$ , y el potencial después es como máximo  $D_n / C 1 / C 2 mH /$ , ya que como máximo  $D_n / C 1$  quedan raíces y no se marca ningún nodo durante la operación. Por lo tanto, el costo amortizado es como máximo

$$\begin{aligned} & O(D_n / C tH / 1) + O(D_n / C 1 / C 2 mH /) \\ & O(D_n / C tH / 1) \end{aligned}$$

ya que podemos escalar las unidades de potencial para dominar la constante oculta en  $O(tH / 1)$ . Intuitivamente, el costo de realizar cada enlace se paga con la reducción del potencial debido a que el enlace reduce el número de raíces en uno. Veremos en la Sección 19.4 que  $D_n / C tH / 1$  es  $O(\lg n)$ , por lo que el costo amortizado de extraer el nodo mínimo es  $O(\lg n)$ .

### Ejercicios

#### 19.2-1

Muestre el montón de Fibonacci que resulta de llamar a FIB-HEAP-EXTRACT-MIN en el montón de Fibonacci que se muestra en la Figura 19.4(m).

### 19.3 Disminución de una clave y eliminación de un nodo

En esta sección, mostramos cómo disminuir la clave de un nodo en un montón de Fibonacci en  $O(1)$  tiempo amortizado y cómo eliminar cualquier nodo de un montón de Fibonacci de  $n$  nodos en  $O(n)$  tiempo amortizado. En la Sección 19.4, mostraremos que la máxima

El grado mínimo  $D_n$  es  $O.\lg n$ , lo que implicará que FIB-HEAP-EXTRACT-MIN y FIB-HEAP-DELETE se ejecutan en  $O.\lg n$  tiempo amortizado.

#### Disminución de una clave

En el siguiente pseudocódigo para la operación FIB-HEAP-DECREASE-KEY, asumimos como antes que eliminar un nodo de una lista enlazada no cambia ninguno de los atributos estructurales del nodo eliminado.

FIB-MONTÓN-DISMINUCIÓN-TECLA.H; X; k/

```

1 si k > x:key error
    "la nueva key es mayor que la actual" 2 3 x:key D k
4 y D x:p 5 si y ≠
    NIL y x:key <
    y:key 6 CUT.H; X; y/ 7 CORTE EN
    CASCADA.H; y/ 8 si x:tecla
    < H:min:tecla 9 H:min D x

```

CORTE.H; X; y/

1 elimina x de la lista secundaria de y, disminuyendo y: grado 2 agrega x  
a la lista raíz de H

3 x:p D NIL 4  
x:marca D FALSO

CORTE EN CASCADA.H; y/ 1

```

' D y:p 2 if ' ≠
NIL 3 if y:mark =
= FALSE 4 y:mark D TRUE else
CUT.H; y; '/ CORTE EN CASCADA.H;
'
5 6

```

El procedimiento FIB-HEAP-DECREASE-KEY funciona de la siguiente manera. Las líneas 1 a 3 aseguran que la nueva clave no sea mayor que la clave actual de x y luego asignan la nueva clave a x. Si x es una raíz o si x:key y:key, donde y es el padre de x, entonces no es necesario que se produzcan cambios estructurales, ya que no se ha violado el orden del montón mínimo. Las líneas 4 y 5 prueban esta condición.

Si se ha violado el orden mínimo del montón, pueden ocurrir muchos cambios. Comenzamos cortando x en la línea 6. El procedimiento CUT “corta” el vínculo entre x y su padre y, convirtiendo a x en raíz.

Usamos los atributos de marca para obtener los límites de tiempo deseados. Registran un pedacito de la historia de cada nodo. Suponga que los siguientes eventos han ocurrido en el nodo x:

1. en algún momento, x fue una raíz,
2. entonces x se vinculó a (hizo hijo de) otro nodo,
3. luego se sacaron dos hijos de x por cortes.

Tan pronto como se pierde el segundo hijo, cortamos x de su padre, convirtiéndolo en una nueva raíz. El atributo x:mark es VERDADERO si se han realizado los pasos 1 y 2 y se ha cortado un elemento secundario de x. El procedimiento CUT , por lo tanto, borra x:mark en la línea 4, ya que realiza el paso 1. (Ahora podemos ver por qué la línea 3 de FIB-HEAP-LINK borra y:mark: el nodo y está siendo vinculado a otro nodo, y así se está realizando el paso 2. La próxima vez que se corte un elemento secundario de y, y:mark se establecerá en TRUE).

Todavía no hemos terminado, porque x podría ser el segundo hijo cortado de su padre y desde el momento en que y se vinculó a otro nodo. Por lo tanto, la línea 7 de FIB-HEAP DECREASE-KEY intenta realizar una operación de corte en cascada en y. Si y es una raíz, entonces la prueba en la línea 2 de CASCADING-CUT hace que el procedimiento regrese.

Si y no está marcada, el procedimiento la marca en la línea 4, ya que acaba de cortar su primer hijo, y regresa. Sin embargo, si se marca y, acaba de perder a su segundo hijo; y se corta en la línea 5, y CASCADING-CUT se llama a sí mismo recursivamente en la línea 6 en el parente ' de y. El procedimiento CASCADING-CUT recorre su camino hacia arriba en el árbol hasta que encuentra una raíz o un nodo sin marcar.

Una vez que se han producido todos los cortes en cascada, las líneas 8 y 9 de FIB-HEAP-DECREASE KEY terminan actualizando H:min si es necesario. El único nodo cuya clave cambió fue el nodo x cuya clave disminuyó. Por tanto, el nuevo nodo mínimo es el nodo mínimo original o el nodo x.

La figura 19.5 muestra la ejecución de dos llamadas de FIB-HEAP-DECREASE-KEY, comenzando con el montón de Fibonacci que se muestra en la figura 19.5(a). La primera llamada, que se muestra en la Figura 19.5(b), no implica cortes en cascada. La segunda llamada, que se muestra en las Figuras 19.5(c)–(e), invoca dos cortes en cascada.

Ahora mostraremos que el costo amortizado de FIB-HEAP-DECREASE-KEY es solo O.1/. Comenzamos por determinar su costo real. El procedimiento FIB-HEAP-DECREASE KEY toma 0.1/ tiempo, más el tiempo para realizar los cortes en cascada. Suponga que una invocación determinada de FIB-HEAP-DECREASE-KEY da como resultado c llamadas de CASCADING-CUT (la llamada realizada desde la línea 7 de FIB-HEAP-DECREASE-KEY seguida de c 1 llamadas recursivas de CASCADING-CUT). Cada llamada de CORTE EN CASCADA toma 0.1/ tiempo sin contar las llamadas recursivas. Por lo tanto, el costo real de FIB HEAP-DECREASE-KEY, incluidas todas las llamadas recursivas, es Oc/.

A continuación calculamos el cambio de potencial. Sea H el montón de Fibonacci justo antes de la operación FIB-HEAP-DECREASE-KEY . La llamada a CUT en la línea 6 de

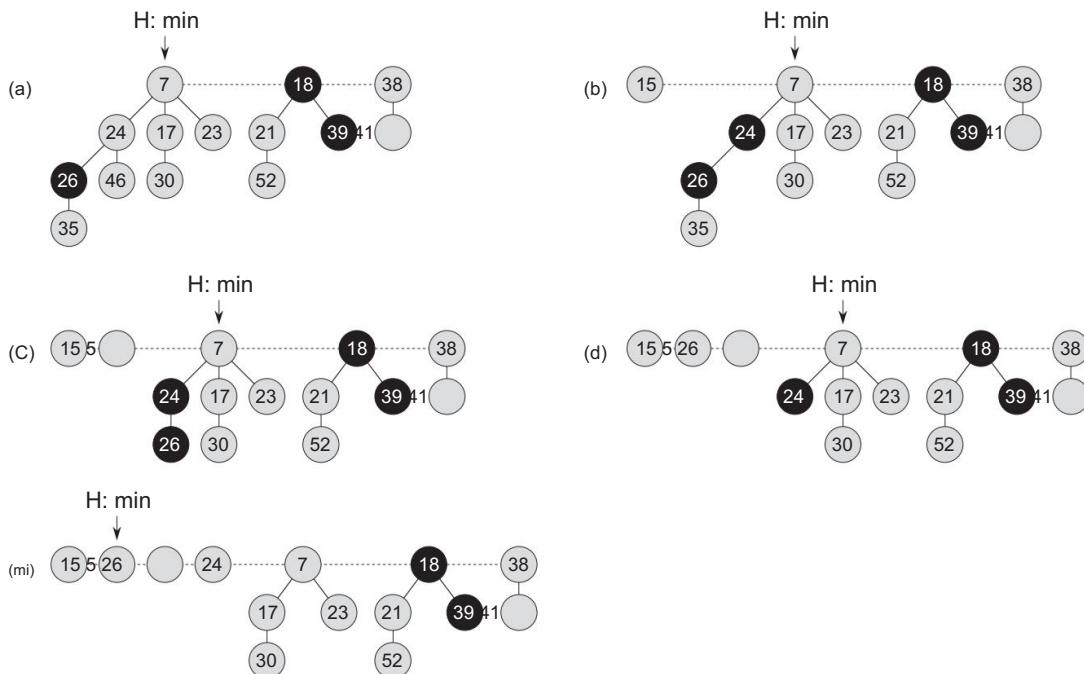


Figura 19.5 Dos llamadas de FIB-HEAP-DECREASE-KEY. (a) El montón inicial de Fibonacci. (b) El nodo con la clave 46 tiene su clave reducida a 15. El nodo se convierte en raíz y su padre (con la clave 24), que anteriormente no estaba marcado, se vuelve marcado. (c)–(e) El nodo con clave 35 tiene su clave reducida a 5. En la parte (c), el nodo, ahora con clave 5, se convierte en raíz. Su padre, con la clave 26, está marcado, por lo que se produce un corte en cascada. El nodo con clave 26 se corta de su padre y se convierte en una raíz sin marcar en (d). Se produce otro corte en cascada, ya que también se marca el nodo con la clave 24.

Este nodo se corta de su padre y se convierte en una raíz sin marcar en la parte (e). Los cortes en cascada se detienen en este punto, ya que el nodo con clave 7 es una raíz. (Incluso si este nodo no fuera una raíz, los cortes en cascada se detendrían, ya que no está marcado). La parte (e) muestra el resultado de la operación FIB-HEAP-DECREASE-KEY , con H:min apuntando al nuevo nodo mínimo .

FIB-HEAP-DECREASE-KEY crea un nuevo árbol enraizado en el nodo  $x$  y borra el bit de marca de  $x$  (que puede haber sido FALSO). Cada llamada de CASCADING-CUT, excepto la última, corta un nodo marcado y borra el bit de marca. Posteriormente, el montón de Fibonacci contiene árboles  $tH / Cc$  (los árboles  $tH /$  originales, los árboles  $c1$  producidos por cortes en cascada y el árbol arraigado en  $x$ ) y como máximo  $mH / cC2$  nodos marcados ( $c1$  no estaban marcados por cortes en cascada y el último llamada de CORTE EN CASCADA puede haber marcado un nodo). Por lo tanto, el cambio de potencial es a lo sumo

$\dots tH / C c / C 2.mH / c C 2 // \dots tH / C 2 mH // D 4 c$

Así, el coste amortizado de FIB-HEAP-DECREASE-KEY es como máximo

Oc/ C 4 c D O.1/ ;

ya que podemos escalar las unidades de potencial para dominar la constante oculta en Oc/.

Ahora puede ver por qué definimos la función potencial para incluir un término que es el doble del número de nodos marcados. Cuando un nodo marcado y es cortado por un corte en cascada, su bit de marca se borra, lo que reduce el potencial en 2. Una unidad de potencial paga por el corte y la limpieza del bit de marca, y la otra unidad compensa el aumento de la unidad en potencial debido a que el nodo y se convierte en raíz.

#### Eliminación de un nodo

El siguiente pseudocódigo elimina un nodo de un montón de Fibonacci de n nodos en ODn// tiempo amortizado. Suponemos que no hay un valor clave de 1 actualmente en el montón de Fibonacci.

FIB-MONTÓN-ELIMINAR.H;

x/ 1 FIB-HEAP-DECREASE-KEY.H; X; 1/ 2 FIB-  
HEAP-EXTRACT-MIN.H /

FIB-HEAP-DELETE hace que x se convierta en el nodo mínimo en el montón de Fibonacci al darle una clave pequeña única de 1. El procedimiento FIB-HEAP-EXTRACT-MIN luego elimina el nodo x del montón de Fibonacci. El tiempo amortizado de FIB-HEAP DELETE es la suma del O.1/ tiempo amortizado de FIB-HEAP-DECREASE-KEY y el ODn// tiempo amortizado de FIB-HEAP-EXTRACT-MIN. Como veremos en la Sección 19.4 que Dn/ D O.lg n/, el tiempo amortizado de FIB-HEAP-DELETE es O.lg n/.

#### Ejercicios

19.3-1

Suponga que se marca una raíz x en un montón de Fibonacci. Explique cómo x llegó a ser una raíz marcada. Argumente que no importa para el análisis que x esté marcada, aunque no sea una raíz que primero se vinculó a otro nodo y luego perdió un hijo.

19.3-2

Justificar el O.1/ tiempo amortizado de FIB-HEAP-DECREASE-KEY como costo promedio por operación utilizando análisis agregado.

## 19.4 Acotando el grado máximo

Para demostrar que el tiempo amortizado de FIB-HEAP-EXTRACT-MIN y FIB-HEAP-DELETE es  $O(\lg n)$ , debemos demostrar que el límite superior  $D_n$  en el grado de cualquier nodo de un montón de Fibonacci de  $n$  nodos es  $O(\lg n)$ . En particular, mostraremos que  $D_n / \log n \leq \phi$ , donde  $\phi$  es la proporción áurea, definida en la ecuación (3.24) como

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61803 \dots$$

La clave del análisis es la siguiente. Para cada nodo  $x$  dentro de un montón de Fibonacci, defina  $\text{size}(x)$  para que sea el número de nodos, incluido el propio  $x$ , en el subárbol con raíz en  $x$ . (Tenga en cuenta que  $x$  no necesita estar en la lista raíz; puede ser cualquier nodo). Mostraremos que  $\text{size}(x)$  es exponencial en  $x:\text{grado}$ . Tenga en cuenta que  $x:\text{grado}$  siempre se mantiene como un recuento exacto del grado de  $x$ .

### Lema 19.1

Sea  $x$  cualquier nodo en un montón de Fibonacci, y suponga que  $x:\text{grado} \leq D$ . Sea  $y_1; y_2; \dots; y_k$  denota a los hijos de  $x$  en el orden en que fueron vinculados a  $x$ , del más antiguo al más reciente. Entonces,  $y_i:\text{grado} \leq D$  para  $i = 1, 2, \dots, k$ .

Prueba Obviamente,  $y_1:\text{grado} = 0$ .

para mí      2, observamos que cuando  $y_i$  se vinculaba con  $x$ , todo  $y_1; y_2; \dots; y_i$  eran hijos de  $x$ , por lo que debemos haber tenido  $x:\text{grado} \geq i+1$ . Debido a que el nodo  $y_i$  está vinculado a  $x$  (por CONSOLIDAR) solo si  $x:\text{grado} \geq y_i:\text{grado}$ , también debemos haber tenido  $y_i:\text{grado} \geq 1$  en ese momento. Desde entonces, el nodo  $y_i$  ha perdido como máximo un hijo, ya que habría sido cortado de  $x$  (por CORTE EN CASCADA) si hubiera perdido dos hijos. Concluimos que  $y_i:\text{grado} \leq i$ . ■

Finalmente llegamos a la parte del análisis que explica el nombre de “montones de Fibonacci”. Recuerde de la Sección 3.2 que para  $k \geq 0$ , el  $k$ -ésimo número de Fibonacci se define por la recurrencia

$$\begin{aligned} F_k &= 0 & \text{si } k = 0 \\ F_k &= 1 & \text{si } k = 1 \\ F_k &= F_{k-1} + F_{k-2} & \text{si } k \geq 2 \end{aligned}$$

El siguiente lema da otra forma de expresar  $F_k$ .

## Lema 19.2

Para todos los enteros  $k \geq 0$ ,

$$\sum_{i=0}^k F_i = F_{k+1}$$

Demostración La demostración es por inducción sobre  $k$ . Cuando  $k = 0$ ,

$$\begin{aligned} & 0 \\ 1 & CX \quad F_0 = 1 \\ & D 1 C 0 \\ & D F_2 : \end{aligned}$$

Ahora asumimos la hipótesis inductiva de que  $\sum_{i=0}^{k-1} F_i = F_k$ , y nosotros tener

$$\begin{aligned} & F_k + \sum_{i=0}^{k-1} F_i = F_k + F_k \\ & D F_k C 1 CX \quad F_k + F_k = F_{k+1} \\ & D 1 CX \quad F_k + F_k = F_{k+1} : \end{aligned}$$

## Lema 19.3

Para todos los enteros  $k \geq 0$ , el número de Fibonacci  $F_k$  satisface  $F_k \leq \frac{1}{\sqrt{5}}(1 + \sqrt{5})^k$ .

Demostración La demostración es por inducción sobre  $k$ . Los casos base son para  $k = 0$  y  $k = 1$ . Cuando  $k = 0$  tenemos  $F_0 = 1 \leq \frac{1}{\sqrt{5}}(1 + \sqrt{5})^0$ , y cuando  $k = 1$  tenemos  $F_1 = 1 \leq \frac{1}{\sqrt{5}}(1 + \sqrt{5})^1$ . El paso inductivo es para  $k = 2$ , y suponemos que  $F_k \leq \frac{1}{\sqrt{5}}(1 + \sqrt{5})^k$ . Por la ecuación (3.23), tenemos

$$\begin{aligned} & F_k + F_{k+1} \leq \frac{1}{\sqrt{5}}(1 + \sqrt{5})^k + \frac{1}{\sqrt{5}}(1 + \sqrt{5})^{k+1} \\ & D F_k C 1 CX \quad F_k + F_{k+1} \leq \frac{1}{\sqrt{5}}(1 + \sqrt{5})^{k+1} \\ & D F_{k+1} C 1 CX \quad F_{k+1} \leq \frac{1}{\sqrt{5}}(1 + \sqrt{5})^{k+1} \end{aligned}$$

El siguiente lema y su corolario completan el análisis.

## Lema 19.4

Sea  $x$  cualquier nodo en un montón de Fibonacci, y sea  $k \leq D(x)$ : grado. Entonces tamaño. $x / FkC2$  donde  $D(x) \geq 1$  y  $p_5 = 2$ .

Prueba Sea  $sk$  el tamaño mínimo posible de cualquier nodo de grado  $k$  en cualquier montón de Fibonacci. Trivialmente,  $s_0 \leq 1$  y  $s_1 \leq 2$ . El número  $sk$  es como máximo size. $x / y$ , debido a que agregar niños a un nodo no puede disminuir el tamaño del nodo, el valor de  $sk$  aumenta monótonamente con  $k$ . Considere algún nodo  $'$ , en cualquier montón de Fibonacci, tal que  $' : grado \leq k$  y  $tamaño(') \leq sk$ . Debido a que  $sk \leq size.x / y$ , calculamos un límite inferior en size. $x / y$  calculando un límite inferior en  $sk$ . Como en el Lema 19.1, sea  $y_1; y_2; \dots; y_k$  denotan los hijos de  $'$  en el orden en que estaban vinculados a  $'$ . Para acotar  $sk$ , contamos uno para  $'$  y uno para el primer hijo  $y_1$  (para el cual  $size.y_1 / 1$ ), dando

$$\begin{aligned} \text{tamaño.}x / & \quad sk \\ & \quad k \\ 2CX_{iD2} & \quad \text{si } i : \text{grado} \\ & \quad k \\ 2CX_{iD2} & \quad \text{si } i = 2 ; \end{aligned}$$

donde la última línea se sigue del Lema 19.1 (de modo que  $y_i : \text{grado } i = 2$ ) y la monotonicidad de  $sk$  (de modo que  $\text{si } i : \text{grado} \leq \text{si } j : \text{grado}$ ).

Ahora mostramos por inducción sobre  $k$  que  $sk \leq FkC2$  para todos los enteros no negativos  $k$ . Las bases, para  $k = 0$  y  $k = 1$ , son triviales. Para el paso inductivo, suponemos que  $k \geq 2$  y que si  $FiC2$  para  $i \in \{0, 1, \dots, k\}$ . tenemos

$$\begin{aligned} & \quad k \\ sk & \quad 2CX_{iD2} \quad \text{si } i = 2 \\ & \quad k \\ 2CX_{iD2} & \quad \text{fi} \\ & \quad k \\ D1CX_{iD0} & \quad \text{fi} \\ D FkC2 & \quad (\text{por el Lema 19.2}) \\ & \quad k \\ & \quad (\text{por el Lema 19.3}). \end{aligned}$$

Así, hemos demostrado que  $\text{size.}x / sk \leq FkC2$

$k$ .

■

**Corolario 19.5 El**

grado máximo  $D_n$  de cualquier nodo en un montón de Fibonacci de  $n$  nodos es  $O.\lg n$ .

Prueba Sea  $x$  cualquier nodo en un montón de Fibonacci de  $n$  nodos, y sea  $k$   $D_x$ : grado.  $k$ .

19.4, tenemos  $n$  tamaño. $x / \log n^{\sqrt{}}.$ ) El máximo Tomando logaritmos base da Por el Lema número entero,  $k$  grado  $D_n$  de cualquier nodo ~~es menor que~~ De hecho, debido a que  $k$  es un  $O.\lg n$ . ■

**Ejercicios****19.4-1**

El profesor Pinocho afirma que la altura de un montón de Fibonacci de  $n$  nodos es  $O.\lg n$ .

Muestre que el profesor se equivoca al exhibir, para cualquier número entero positivo  $n$ , una secuencia de operaciones de montón de Fibonacci que crea un montón de Fibonacci que consta de un solo árbol que es una cadena lineal de  $n$  nodos.

**19.4-2**

Supongamos que generalizamos la regla del corte en cascada para cortar un nodo  $x$  de su padre tan pronto como pierda su  $k$ -ésimo hijo, para alguna constante entera  $k$ . (La regla de la sección 19.3 usa  $k = D_2$ .) ¿Para qué valores de  $k$  es  $D_n = O.\lg n$ ?

**Problemas****19-1 Implementación alternativa de la eliminación El**

profesor Pisano ha propuesto la siguiente variante del procedimiento FIB-HEAP-DELETE , alegando que se ejecuta más rápido cuando el nodo que se elimina no es el nodo señalado por  $H:\min$ .

PISANO-ELIMINAR.H;  $x / 1$

si  $x == H:\min$  FIB-

HEAP-EXTRACT-MIN.H / 2 3 más y

$D_x:p 4$  si  $y \neq NIL$

5 CUT.H;  $X; y / 6$

CORTE EN CASCADA.H;  $y / 7$

8

agregue la lista secundaria de  $x$  a la lista

raíz de  $H$  elimine  $x$  de la lista raíz de  $H$

- a. La afirmación del profesor de que este procedimiento es más rápido se basa en parte en la suposición de que la línea 7 se puede realizar en O.1/ en tiempo real. ¿Qué hay de malo en esta suposición?
- b. Proporcione un buen límite superior para el tiempo real de PISANO-DELETE cuando  $x$  no es H:min. Su límite debe ser en términos de  $x$ :grado y el número  $c$  de llamadas al procedimiento CASCADING-CUT .
- C. Supongamos que llamamos PISANO-DELETE.H;  $x$ , y sea  $H_0$  el montón de Fibonacci resultante.  
Suponiendo que el nodo  $x$  no es una raíz, acote el potencial de  $H_0$  en términos de  $x$ :grado,  $c$ ,  $tH$  y  $mH$ .
- d. Concluya que el tiempo amortizado para PISANO-DELETE es asintóticamente nulo mejor que para FIB-HEAP-DELETE, incluso cuando  $x \neq$  H:min.

#### 19-2 Árboles binomiales y pilas binomiales El árbol

binomial  $B_k$  es un árbol ordenado (ver Sección B.5.2) definido recursivamente.

Como se muestra en la figura 19.6(a), el árbol binomial  $B_0$  consta de un solo nodo. El árbol binomial  $B_k$  consta de dos árboles binomiales  $B_{k-1}$  que están vinculados entre sí de modo que la raíz de uno es el hijo más a la izquierda de la raíz del otro. La figura 19.6(b) muestra los árboles binomiales  $B_0$  a  $B_4$ .

a. Muestre que para el árbol binomial  $B_k$ ,

1. hay  $2^k$  nodos, 2. la altura

del árbol es  $k$ , 3. hay exactamente  $4^k$

la raíz tiene grado  $k$ , que  $k^i$  nodos en la profundidad  $i$  para  $i \in 0; 1; \dots; k$ , y

es mayor que la de cualquier otro nodo; además, como muestra la figura 19.6(c), si numeramos a los hijos de la raíz de izquierda a derecha por  $k+1; k+2; \dots; 0$ , entonces el niño  $i$  es la raíz de un subárbol  $B_i$ .

Un montón binomial  $H$  es un conjunto de árboles binomiales que satisface las siguientes propiedades:

1. Cada nodo tiene una clave (como un montón de Fibonacci).
2. Cada árbol binomial en  $H$  obedece a la propiedad min-heap.
3. Para cualquier entero no negativo  $k$ , hay al sumo un árbol binomial en  $H$  cuya raíz tiene grado  $k$ .

- b. Suponga que un montón binomial  $H$  tiene un total de  $n$  nodos. Analice la relación entre los árboles binomiales que contiene  $H$  y la representación binaria de  $n$ .  
Concluya que  $H$  consta como máximo de árboles binomiales  $\lg n + C$ .

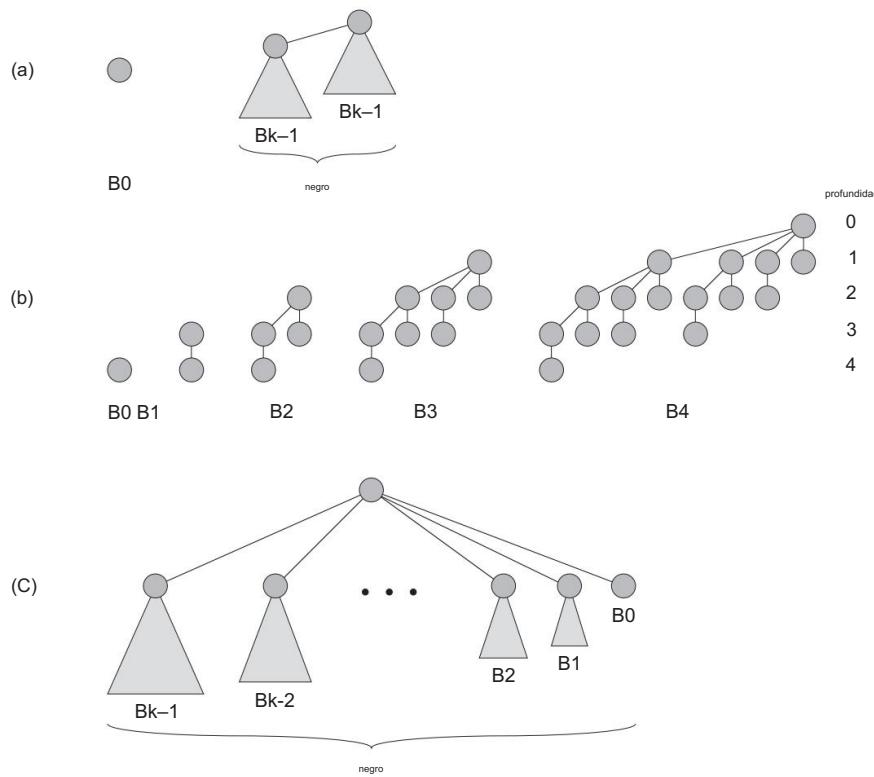


Figura 19.6 (a) La definición recursiva del árbol binomial  $B_k$ . Los triángulos representan subárboles enraizados. (b) Los árboles binomiales  $B_0$  a  $B_4$ . Se muestran las profundidades de los nodos en  $B_4$ . (c) Otra forma de ver el árbol binomial  $B_k$ .

Supongamos que representamos un montón binomial de la siguiente manera. El esquema hijo izquierdo, hermano derecho de la Sección 10.4 representa cada árbol binomial dentro de un montón binomial. Cada nodo contiene su clave; punteros a su padre, a su hijo más a la izquierda y al hermano inmediatamente a su derecha (estos punteros son NIL cuando corresponde); y su grado (como en los montones de Fibonacci, cuántos hijos tiene). Las raíces forman una lista de raíces enlazadas individualmente, ordenadas por los grados de las raíces (de menor a mayor), y accedemos al montón binomial mediante un puntero al primer nodo en la lista de raíces.

C. Complete la descripción de cómo representar un montón binomial (es decir, nombre los atributos, describa cuándo los atributos tienen el valor NIL y defina cómo se organiza la lista raíz) y muestre cómo implementar las mismas siete operaciones en los montones binomiales que en este capítulo. implementado en montones de Fibonacci. Cada operación debe ejecutarse en  $O.\lg n$  en el peor de los casos, donde  $n$  es el número de nodos en

el montón binomial (o en el caso de la operación UNION , en los dos montones binomiales que se están uniendo). La operación MAKE-HEAP debe tomar un tiempo constante.

- d. Supongamos que fuéramos a implementar solo las operaciones de almacenamiento dinámico fusionable en un almacenamiento dinámico de Fibonacci (es decir, no implementamos las operaciones DECREASE-KEY o DELETE ). ¿Cómo se parecerían los árboles en un montón de Fibonacci a los de un montón binomial? ¿Cómo se diferenciarían? Muestre que el grado máximo en un montón de Fibonacci de n nodos sería como máximo  $\lg n$ .

mi. El profesor McGee ha ideado una nueva estructura de datos basada en montones de Fibonacci. Un montón de McGee tiene la misma estructura que un montón de Fibonacci y admite solo las operaciones de montón fusionable. Las implementaciones de las operaciones son las mismas que para los montones de Fibonacci, excepto que la inserción y la unión consolidan la lista raíz como último paso. ¿Cuáles son los peores tiempos de ejecución de las operaciones en los montones de McGee?

#### 19-3 Más operaciones del montón de Fibonacci

Deseamos aumentar un montón H de Fibonacci para admitir dos nuevas operaciones sin cambiar el tiempo de ejecución amortizado de ninguna otra operación del montón de Fibonacci.

- a. La operación FIB-HEAP-CHANGE-KEY.H; X; k/ cambia la clave del nodo x al valor k. Proporcione una implementación eficiente de FIB-HEAP-CHANGE-KEY y analice el tiempo de ejecución amortizado de su implementación para los casos en los que k es mayor, menor o igual que x:key.
- b. Dar una implementación eficiente de FIB-HEAP-PRUNE.H; r/, que suprime q D min.r; H:n/ nodos de H. Puede elegir cualquier nodo q para eliminar. Analice el tiempo de ejecución amortizado de su implementación. (Sugerencia: es posible que deba modificar la estructura de datos y la función potencial).

#### 19-4 2-3-4 montones El

capítulo 18 introdujo el árbol 2-3-4, en el que cada nodo interno (aparte posiblemente de la raíz) tiene dos, tres o cuatro hijos y todas las hojas tienen la misma profundidad. En este problema, implementaremos montones 2-3-4, que admiten las operaciones de montones combinables.

Los montones 2-3-4 difieren de los árboles 2-3-4 de las siguientes maneras. En 2-3-4 montones, solo deja claves de almacenamiento, y cada hoja x almacena exactamente una clave en el atributo x:key. Las claves en las hojas pueden aparecer en cualquier orden. Cada nodo interno x contiene un valor x:pequeño que es igual a la clave más pequeña almacenada en cualquier hoja del subárbol con raíz en x. La raíz r contiene un atributo r:height que da la altura de la

árbol. Finalmente, 2-3-4 montones están diseñados para mantenerse en la memoria principal, de modo que no se necesiten lecturas y escrituras en el disco.

Implemente las siguientes operaciones de montón 2-3-4. En las partes (a)–(e), cada operación debe ejecutarse en  $O.\lg n/$  tiempo en un montón 2-3-4 con  $n$  elementos. La operación UNION de la parte (f) debe ejecutarse en tiempo  $O.\lg n/$ , donde  $n$  es el número de elementos en los dos montones de entrada.

- a. MINIMUM, que devuelve un puntero a la hoja con la clave más pequeña.
  - b. DECREASE-KEY, que disminuye la clave de una hoja dada  $x$  a un valor dado  $k$ : clave.
  - C. INSERT, que inserta la hoja  $x$  con la tecla  $k$ .
  - d. DELETE, que elimina una hoja dada  $x$ .
  - mi. EXTRACT-MIN, que extrae la hoja con la tecla más pequeña.
- F. UNION, que une dos montones 2-3-4, devuelve un único montón 2-3-4 y destruye los montones de entrada.

### Notas del capítulo

Fredman y Tarjan [114] introdujeron los montones de Fibonacci. Su artículo también describe la aplicación de montones de Fibonacci a los problemas de las rutas más cortas de una sola fuente, las rutas más cortas de todos los pares, la coincidencia bipartita ponderada y el problema del árbol de expansión mínima.

Posteriormente, Driscoll, Gabow, Shrairman y Tarjan [96] desarrollaron "montones relajados" como alternativa a los montones de Fibonacci. Idearon dos variedades de montones relajados. Uno da los mismos límites de tiempo amortizados que los montones de Fibonacci. El otro permite que DECREASE-KEY se ejecute en  $O.1/$  en el peor de los casos (no amortizado) y EXTRACT-MIN y DELETE se ejecuten en  $O.\lg n/$  en el peor de los casos. Los montones relajados también tienen algunas ventajas sobre los montones de Fibonacci en algoritmos paralelos.

Consulte también las notas del capítulo 6 para conocer otras estructuras de datos que admiten operaciones rápidas de DECREASE-KEY cuando la secuencia de valores devueltos por las llamadas EXTRACT MIN aumenta de manera monótona con el tiempo y los datos son números enteros en un rango específico.

---

## 20 de Emde Boas árboles

En capítulos anteriores, vimos estructuras de datos que respaldan las operaciones de una cola de prioridad: montones binarios en el Capítulo 6, árboles rojo-negro en el Capítulo 13,<sup>1</sup> y montones de Fibonacci en el Capítulo 19. En cada una de estas estructuras de datos, al menos un importante la operación tomó  $O.lg n$  tiempo, en el peor de los casos o amortizada. De hecho, debido a que cada una de estas estructuras de datos basa sus decisiones en la comparación de claves, el límite inferior  $.n \lg n$  para ordenar en la Sección 8.1 nos dice que al menos una operación tendrá que tomar  $\lg n$  tiempo. ¿Por qué? Si pudiéramos realizar las operaciones INSERT y EXTRACT-MIN en  $O.lg n$  time, entonces podríamos ordenar  $n$  claves en  $\lg n$  time realizando primero  $n$  operaciones INSERT , seguidas de  $n$  operaciones EXTRACT-MIN .

Sin embargo, vimos en el Capítulo 8 que a veces podemos explotar información adicional sobre las claves para clasificar en  $\lg n$  time. En particular, con la ordenación por conteo podemos ordenar  $n$  claves, cada una de ellas un número entero en el rango de 0 a  $k$ , en el tiempo  $.n C k/$ , que es  $.n$  cuando  $k = O(n)$ .

Dado que podemos eludir el límite inferior de  $.n \lg n$  para ordenar cuando las claves son números enteros en un rango limitado, es posible que se pregunte si podemos realizar cada una de las operaciones de cola de prioridad en  $O.lg n$  en un escenario similar. En este capítulo, veremos que podemos: Los árboles de van Emde Boas soportan las operaciones de cola de prioridad, y algunas otras, cada una en  $O.lg \lg n$  en el peor de los casos. El problema es que las claves deben ser números enteros en el rango de 0 a  $n$ , sin duplicados permitidos.

Especificamente, los árboles de van Emde Boas admiten cada una de las operaciones de conjuntos dinámicos enumeradas en la página 230 (BÚSQUEDA, INSERCIÓN, ELIMINACIÓN, MÍNIMO, MÁXIMO, SUCCESSOR y PREDECESOR) en tiempo  $O.lg \lg n$ . En este capítulo, omitiremos la discusión de los datos satelitales y nos centraremos solo en el almacenamiento de claves. Porque nos concentraremos en las claves y no permitimos que se almacenen claves duplicadas, en lugar de describir la BÚSQUEDA

---

<sup>1</sup>El Capítulo 13 no analiza explícitamente cómo implementar EXTRACT-MIN y DECREASE-KEY, pero podemos construir fácilmente estas operaciones para cualquier estructura de datos que admita MINIMUM, DELETE e INSERT.

operación, implementaremos la operación más simple MEMBER.S;  $x/S$ , que devuelve un valor booleano que indica si el valor  $x$  se encuentra actualmente en el conjunto dinámico  $S$ .

Hasta ahora, hemos usado el parámetro  $n$  para dos propósitos distintos: el número de elementos en el conjunto dinámico y el rango de los valores posibles. Para evitar más confusiones, de ahora en adelante usaremos  $n$  para indicar el número de elementos actualmente en el conjunto y  $u$  como el rango de valores posibles, de modo que cada operación de árbol de van Emde Boas se ejecute en  $O.\lg \lg u/t$  tiempo. Llamamos al conjunto  $f_0; 1; 2; \dots; u^{1g}$  el universo de valores que se pueden almacenar y  $u$  el tamaño del universo. Suponemos a lo largo de este capítulo que  $u$  es una potencia exacta de 2, es decir,  $u = 2^k$  para algún número entero  $k \geq 1$ .

La Sección 20.1 comienza examinando algunos enfoques simples que nos llevarán en la dirección correcta. Mejoraremos estos enfoques en la Sección 20.2, introduciendo estructuras proto van Emde Boas, que son recursivas pero no logran nuestro objetivo de operaciones  $O.\lg \lg u/t$ . La Sección 20.3 modifica estructuras proto van Emde Boas para desarrollar árboles van Emde Boas, y muestra cómo implementar cada operación en  $O.\lg \lg u/t$ .

## 20.1 Aproximaciones preliminares

En esta sección, examinaremos varios enfoques para almacenar un conjunto dinámico.

Aunque ninguno alcanzará los límites de tiempo  $O.\lg \lg u/t$  que deseamos, obtendremos conocimientos que nos ayudarán a comprender los árboles de van Emde Boas cuando los veamos más adelante en este capítulo.

### Direccionamiento directo

El direccionamiento directo, como vimos en la Sección 11.1, proporciona el enfoque más simple para almacenar un conjunto dinámico. Dado que en este capítulo solo nos interesa almacenar claves, podemos simplificar el enfoque de direccionamiento directo para almacenar el conjunto dinámico como un vector de bits, como se explica en el ejercicio 11.1-2. Para almacenar un conjunto dinámico de valores del universo  $f_0; 1; 2; \dots; u^{1g}$ , mantenemos un arreglo  $A[0 : : u]$  de  $u$  bits. La entrada  $A[x]$  tiene un 1 si el valor  $x$  está en el conjunto dinámico y un 0 en caso contrario.

Aunque podemos realizar cada una de las operaciones INSERT, DELETE y MEMBER en tiempo  $O.1/t$  con un vector de bits, las operaciones restantes (MINIMUM, MAXIMUM, SUCCESSOR y PREDECESSOR) toman  $\sim u/t$  tiempo en el peor de los casos porque

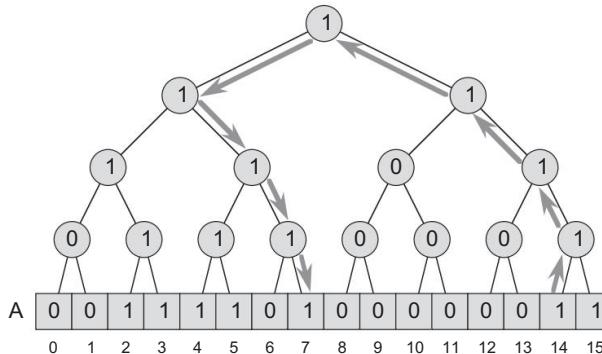


Figura 20.1 Árbol binario de bits superpuestos a un vector de bits que representa el conjunto f2; 3; 4; 5; 7; 14; 15g cuando u D 16. Cada nodo interno contiene un 1 si y solo si alguna hoja en su subárbol contiene un 1. Las flechas muestran el camino seguido para determinar el sucesor de 14 en el vector A.

es posible que tengamos que escanear los elementos „u/2 Por ejemplo, si un conjunto contiene solo los valores 0 y u 1, entonces para encontrar el sucesor de 0, tendríamos que escanear las entradas 1 a u 2 antes de encontrar un 1 en AŒu 1.

#### Superposición de una estructura de árbol binario

Podemos acortar escaneos largos en el vector de bits superponiendo un árbol binario de bits encima. La figura 20.1 muestra un ejemplo. Las entradas del vector de bits forman las hojas del árbol binario, y cada nodo interno contiene un 1 si y solo si alguna hoja en su subárbol contiene un 1. En otras palabras, el bit almacenado en un nodo interno es el lógico-o de sus dos hijos.

Las operaciones que tomaban „u/ en el peor de los casos con un vector de bits sin adornos ahora usan la estructura de árbol:

Para encontrar el valor mínimo en el conjunto, comience en la raíz y diríjase hacia las hojas, siempre tomando el nodo más a la izquierda que contiene un 1.

Para encontrar el valor máximo en el conjunto, comience en la raíz y diríjase hacia las hojas, siempre tomando el nodo más a la derecha que contiene un 1.

<sup>2</sup>Suponemos a lo largo de este capítulo que MINIMUM y MAXIMUM devuelven NIL si el conjunto dinámico está vacío y que SUCCESSOR y PREDECESOR devuelven NIL si el elemento que se les da no tiene sucesor o predecesor, respectivamente.

Para encontrar el sucesor de  $x$ , comience en la hoja indexada por  $x$ , y diríjase hacia la raíz hasta que ingrese un nodo desde la izquierda y este nodo tenga un 1 en su hijo derecho '.

Luego diríjase hacia abajo a través del nodo ', siempre tomando el nodo más a la izquierda que contiene un 1 (es decir, encuentre el valor mínimo en el subárbol con raíz en el hijo derecho ').

Para encontrar el predecesor de  $x$ , comience en la hoja indexada por  $x$ , y diríjase hacia la raíz hasta que ingrese un nodo desde la derecha y este nodo tenga un 1 en su hijo izquierdo '. Luego diríjase hacia abajo a través del nodo ', siempre tomando el nodo más a la derecha que contiene un 1 (es decir, encuentre el valor máximo en el subárbol con raíz en el hijo izquierdo ').

La figura 20.1 muestra el camino seguido para encontrar el predecesor, 7, del valor 14.

También aumentamos las operaciones INSERT y DELETE apropiadamente. Cuando ingresamos un valor, almacenamos un 1 en cada nodo en el camino simple desde la hoja apropiada hasta la raíz. Al eliminar un valor, vamos desde la hoja correspondiente hasta la raíz, recalculando el bit en cada nodo interno del camino como el o lógico de sus dos hijos.

Dado que la altura del árbol es  $\lg u$  y cada una de las operaciones anteriores hace como máximo una pasada por el árbol y como máximo una pasada hacia abajo, cada operación lleva  $O(\lg u)$  de tiempo en el peor de los casos.

Este enfoque es solo marginalmente mejor que simplemente usar un árbol rojo-negro. Todavía podemos realizar la operación MEMBER en tiempo  $O(1)$ , mientras que buscar un árbol rojo-negro toma  $O(\lg n)$  tiempo. Por otra parte, si el número  $n$  de elementos almacenados es mucho menor que el tamaño  $u$  del universo, un árbol rojo-negro sería más rápido para todas las demás operaciones.

### Superposición de un árbol de altura constante

¿Qué pasa si superponemos un árbol con mayor grado? Supongamos que el tamaño del universo es  $u = 2^{2k}$  para algún número entero  $k$ , por lo que  $p_u$  es un número entero.

En lugar de superponer un árbol binario encima del vector de bits, superponemos un árbol de grado  $p_u$ . La figura 20.2(a) muestra un árbol de este tipo para el mismo vector de bits que en la figura 20.1. La altura del árbol resultante es siempre 2.

Como antes, cada nodo interno almacena el o lógico de los bits dentro de su subárbol, de modo que los nodos internos  $p_u$  en la profundidad 1 resumen cada grupo de valores  $p_u$ . Como demuestra la figura 20.2(b), podemos pensar en estos nodos como un arreglo  $\text{summary}[0 : p_u - 1]$ , donde  $\text{summary}[i]$  contiene un 1 si y solo si el subrayo  $A[i:p_u] : : .i C[1:p_u - 1]$  contiene un 1. Llame a este subarreglo  $p_u$ -bit de  $A$  el grupo  $i$ . Para un valor dado de  $x$ , el bit  $A[x]$  aparece en el grupo número  $b_x = p_u$ . Ahora INSERT se convierte en una operación  $O(1)$ -time: para insertar  $x$ , establezca  $A[x]$  y  $\text{summary}[b_x] = 1$ . Podemos usar la matriz de resumen para realizar

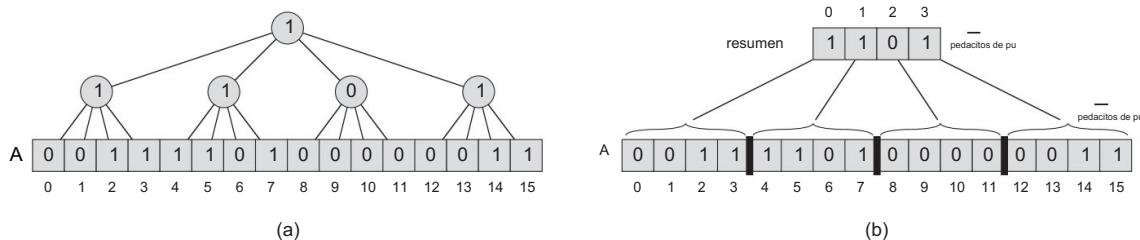


Figura 20.2 (a) Un árbol de grado pu superpuesto sobre el mismo vector de bits que en la Figura 20.1. Cada nodo interno almacena el o lógico de los bits en su subárbol. (b) Una vista de la misma estructura, pero con los nodos internos en la profundidad 1 tratados como un arreglo summary $\overline{E}_0 : : pu 1$ , donde summary $\overline{E}_i$  es el o lógico del subarreglo  $A[\overline{C}_i : : \overline{T}_i]$ .

cada una de las operaciones MINIMO, MAXIMO, SUCESOR, PREDECESOR y ELIMINAR en tiempo O.pu/:

Para encontrar el valor mínimo (máximo), busque la entrada más a la izquierda (más a la derecha) en el resumen que contiene un 1, por ejemplo, summary $\overline{E}_i$ , y luego realice una búsqueda lineal dentro del  $i$ -ésimo grupo para el 1 más a la izquierda (más a la derecha).

Para encontrar el sucesor (predecesor) de  $x$ , primero busque a la derecha (izquierda) dentro de su grupo. Si encontramos un 1, esa posición da el resultado. De lo contrario, sea  $i \leq bx = puc$  y busque a la derecha (izquierda) dentro de la matriz de resumen del índice  $i$ . La primera posición que contiene un 1 da el índice de un grupo. Busque dentro de ese grupo el 1 más a la izquierda (más a la derecha). Esa posición contiene al sucesor (predecesor).

Para borrar el valor  $x$ , sea  $i \leq bx = puc$ . Establezca  $A[\overline{C}_i : : \overline{T}_i]$  en 0 y luego establezca summary $\overline{E}_i$  en el o lógico de los bits en el grupo  $i$ .

En cada una de las operaciones anteriores, buscamos a lo sumo dos grupos de bits pu más la matriz de resumen, por lo que cada operación requiere tiempo O.pu/.

A primera vista, parece que hemos hecho un progreso negativo. La superposición de un árbol binario nos dio operaciones  $O.lg u$ -time, que son asintóticamente más rápidas que  $O.pu/$  time. Sin embargo, usar un árbol de grado pu resultará ser una idea clave de los árboles de van Emde Boas. Continuaremos por este camino en la siguiente sección.

## Ejercicios

### 20.1-1

Modifique las estructuras de datos en esta sección para admitir claves duplicadas.

## 20.1-2

Modifique las estructuras de datos en esta sección para admitir claves que tengan datos de satélite asociados.

## 20.1-3

Observe que, utilizando las estructuras de esta sección, la forma en que encontramos el sucesor y el predecesor de un valor  $x$  no depende de si  $x$  está en el conjunto en ese momento.

Muestre cómo encontrar el sucesor de  $x$  en un árbol de búsqueda binario cuando  $x$  no está almacenado en el árbol.

## 20.1-4

Suponga que en lugar de superponer un árbol de grado  $p_u$ , superpusiéramos un árbol de grado  $u_1=k$ , donde  $k>1$  es una constante. ¿Cuál sería la altura de tal árbol y cuánto tiempo tomaría cada una de las operaciones?

## 20.2 Una estructura recursiva

En esta sección, modificamos la idea de superponer un árbol de grado  $p_u$  encima de un vector de bits. En la sección anterior, usamos una estructura de resumen de tamaño  $p_u$ , con cada entrada apuntando a otra estructura de tamaño  $p_u$ . Ahora, hacemos que la estructura sea recursiva, reduciendo el tamaño del universo por la raíz cuadrada en cada nivel de recursividad.

Comenzando con un universo de tamaño  $u$ , creamos estructuras que contienen  $p_u$  elementos, que a su vez contienen estructuras de  $u_1=4$  elementos, que contienen estructuras de  $u_1=8$  elementos, y así sucesivamente, hasta un tamaño base de 2.

Para simplificar, en esta sección supondremos que  $u \geq 2^{2k}$  para algún entero  $k$ , de modo que  $u; u_1=2; u_1=4; \dots$  son números enteros. Esta restricción sería bastante severa en la práctica, permitiendo solo valores de  $u$  en la secuencia  $2; 4; 16; 64; 256; 1024; \dots$ . Veremos en la sección cómo relajar esta suposición y suponer solo que  $u \geq 2^k$  para algún número entero  $k$ . Dado que la estructura que examinamos en esta sección es solo un precursor de la verdadera estructura de árbol de van Emde Boas, toleraremos esta restricción a favor de ayudar a nuestra comprensión.

Recordando que nuestro objetivo es lograr tiempos de ejecución de  $O(\lg \lg u)$  para las operaciones, pensemos en cómo podríamos obtener tales tiempos de ejecución. Al final de la Sección 4.3, vimos que al cambiar las variables, podíamos demostrar que la recurrencia

$$T(n) \leq 2T(p_n) + C \lg n \quad (20.1)$$

tiene la solución  $T(n) \leq O(\lg n \lg \lg n)$ . Consideraremos una similar, pero más simple, reaparición:

$$T(u) \leq DT(p_u) + C \lg u \quad (20.2)$$

Si usamos la misma técnica, cambiando variables, podemos mostrar que la recurrencia (20.2) tiene la solución  $T .u/ D O.lg lg u/$ . Sea  $m D lg u$ , de modo que  $u D 2m$  y tenemos

$T .2m/ DT .2m=2/ CO.1/ :$

Ahora renombramos  $Sm/ DT .2m/$ , dando la nueva recurrencia

$Sm/ D Sm=2/ C O.1/ :$

Por el caso 2 del método maestro, esta recurrencia tiene como solución  $Sm/ D O.lg m/$ . Volvemos a cambiar de  $Sm/$  a  $T .u/$ , dando  $T .u/ DT .2m/ D Sm/ D O.lg m/ D O.lg lg u/$ .

La recurrencia (20.2) guiará nuestra búsqueda de una estructura de datos. Diseñaremos una estructura de datos recursiva que se contrae por un factor de  $\frac{1}{2}$  en cada nivel de su recursividad. Cuando una operación atraviesa esta estructura de datos, pasará una cantidad de tiempo constante en cada nivel antes de recurrir al nivel inferior. La recurrencia (20.2) caracterizará entonces el tiempo de ejecución de la operación.

Aquí hay otra manera de pensar cómo el término  $lg lg u$  termina en la solución de recurrencia (20.2). Cuando observamos el tamaño del universo en cada nivel de la estructura de datos recursiva, vemos la secuencia  $u; u_1=2; u_1=4; u_1=8; \dots$ . Si consideramos cuántos bits necesitamos para almacenar el tamaño del universo en cada nivel, necesitamos  $lg u$  en el nivel superior, y cada nivel necesita la mitad de los bits del nivel anterior. En general, si comenzamos con  $b$  bits y reducimos a la mitad el número de bits en cada nivel, luego de los niveles de  $lg b$ , bajamos a solo un bit. Dado que  $b D lg u$ , vemos que después de los niveles de  $lg lg u$ , tenemos un tamaño de universo de 2.

Mirando hacia atrás en la estructura de datos de la Figura 20.2, un valor dado  $x$  reside en el número de clúster  $bx=puc$ . Si consideramos  $x$  como un entero binario  $lg u$ -bit, ese número de grupo,  $bx=puc$ , viene dado por los  $.lg u/2$  bits más significativos de  $x$ . Dentro de su grupo,  $x$  aparece en la posición  $x \bmod pu$ , que viene dada por el menos significativo  $.lg u/2$  bits de  $x$ . Tendremos que indexar de esta manera, así que definamos algunas funciones que nos ayudarán a hacerlo:

alto.x/ D  $x=puc^{\wedge}$ ; bajo.x/  $D x$

$\bmod pu ; \text{índice}.x; y/ D x \bmod pu^{\wedge} C$

y :

La función  $high.x/$  da los  $.lg u/2$  bits más significativos de  $x$ , produciendo el número de grupos de  $x$ . La función  $low.x/$  proporciona los  $.lg u/2$  bits menos significativos de  $x$  y proporciona la posición de  $x$  dentro de su grupo. La función  $index.x; y/$  construye un número de elemento a partir de  $x$ , tratando a  $x$  como los  $.lg u/2$  bits más significativos del número de elemento ya y como los  $.lg u/2$  bits menos significativos. Tenemos la identidad  $x D index.high.x/; bajo.x//$ . El valor de  $u$  utilizado por cada una de estas funciones será

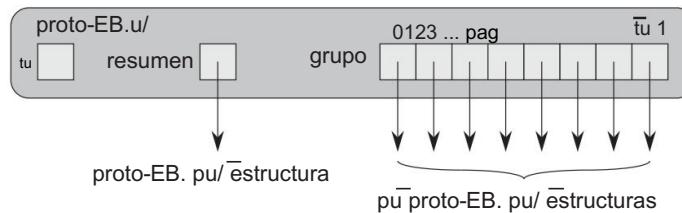


Figura 20.3 La información en una estructura proto-EB.u/ cuando u 4. La estructura contiene el tamaño del universo u, un resumen de puntero a un proto-EB. estructura pu/, y un arreglo cluster0 : : pu de punteros a proto-EB. pu/ estructuras.

siempre será el tamaño del universo de la estructura de datos en la que llamamos a la función, que cambia a medida que descendemos a la estructura recursiva.

#### 20.2.1 Estructuras Proto van Emde Boas

Siguiendo el ejemplo de la recursividad (20.2), diseñemos una estructura de datos recursiva para respaldar las operaciones. Aunque esta estructura de datos no logrará nuestro objetivo de  $O(\lg \lg u)$  tiempo para algunas operaciones, sirve como base para la estructura de árbol de van Emde Boas que veremos en la Sección 20.3.

Para el universo  $f_0; 1; 2; \dots; u^{1g}$ , definimos una estructura proto van Emde Boas, o estructura proto-vEB, que denotamos como proto-EB.u/, recursivamente como sigue. Cada estructura proto-EB.u/ contiene un atributo u que da el tamaño de su universo. Además, contiene lo siguiente:

Si  $u \leq 2^k$ , entonces es el tamaño base y contiene una matriz  $A \in \{0, 1\}^{u \times 1}$ .

De lo contrario,  $u > 2^k$  para algún entero  $k \geq 1$ , de modo que  $u = 2^k + x$ . Además del tamaño del universo u, la estructura de datos proto-EB.u/ contiene los siguientes atributos, ilustrados en la figura 20.3:

un puntero llamado resumen a un proto-EB. pu/ estructura y un arreglo cluster0 : : pu de punteros pu, cada uno a un proto-EB. pu/ estructura

El elemento  $x$ , donde  $0 \leq x < u$ , se almacena recursivamente en el grupo numerado alto.x/ como elemento bajo.x/ dentro de ese grupo.

En la estructura de dos niveles de la sección anterior, cada nodo almacena una matriz de resumen de tamaño pu, en la que cada entrada contiene un bit. A partir del índice de cada entrada, podemos calcular el índice inicial del subarray de tamaño pu que resume el bit. En la estructura proto-vEB, usamos punteros explícitos en lugar de índice

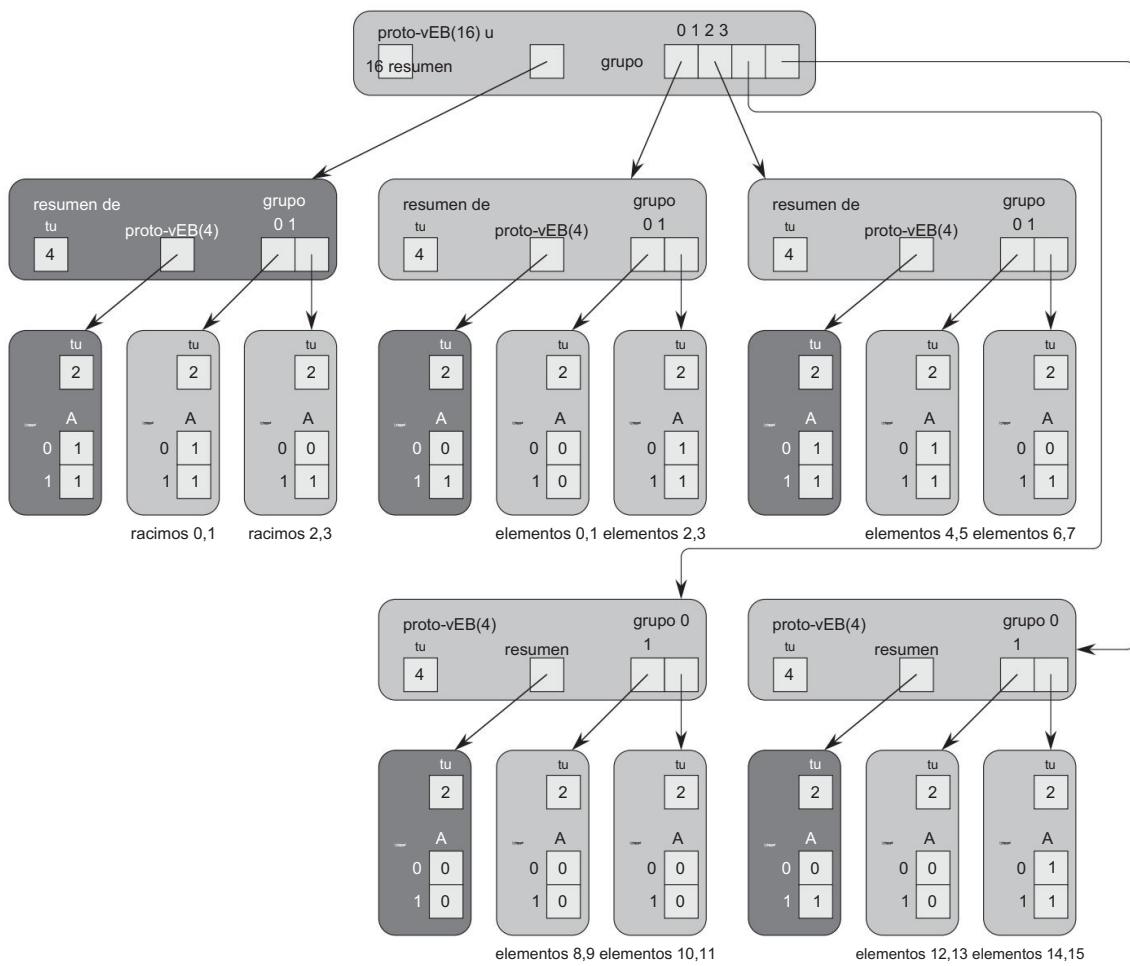


Figura 20.4 Una estructura proto-EB.16/ que representa el conjunto f2; 3; 4; 5; 7; 14; 15 g. Apunta a cuatro estructuras proto-EB.4/ en cluster $\in$ 0 : : 3, ya una estructura resumen, que también es un proto-EB.4/.

Cada estructura proto-EB.4/ apunta a dos estructuras proto-EB.2/ en cluster $\in$ 0 : : 1, y a un resumen proto-EB.2/. Cada estructura proto-EB.2/ contiene solo una matriz A $\in$ 0 : : 1 de dos bits.

Las estructuras proto-EB.2/ por encima de los "elementos i,j" almacenan los bits i y j del conjunto dinámico real, y las estructuras proto-EB.2/ por encima de los "clusters i,j" almacenan los bits de resumen para los clusters i y j en la estructura proto-EB.16/ de nivel superior . Para mayor claridad, el sombreado intenso indica el nivel superior de una estructura proto-vEB que almacena información de resumen para su estructura principal; por lo demás, dicha estructura proto-vEB es idéntica a cualquier otra estructura proto-vEB con el mismo tamaño de universo.

calculos El resumen de la matriz contiene los bits de resumen almacenados recursivamente en una estructura proto-vEB, y el clúster de la matriz contiene punteros pu.

La figura 20.4 muestra una estructura proto-EB.16/ completamente expandida que representa el conjunto  $f_2; 3; 4; 5; 7; 14; 15$ . Si el valor i está en la estructura proto-vEB señalada por resumen, entonces el grupo i-ésimo contiene algún valor en el conjunto que se representa. Como en el árbol de altura constante, cluster $\langle\!\rangle_i$  representa los valores i pu a través de .i C 1/ pu 1, que forman el i-ésimo grupo.

En el nivel base, los elementos de los conjuntos dinámicos reales se almacenan en algunas de las estructuras proto-EB.2/, y las estructuras proto-EB.2/ restantes almacenan bits de resumen. Debajo de cada una de las estructuras base no resumidas, la figura indica qué bits almacena. Por ejemplo, la estructura proto-EB.2/ etiquetada como "elementos 6,7" almacena el bit 6 (0, ya que el elemento 6 no está en el conjunto) en su A $\langle\!\rangle_0$  y el bit 7 (1, ya que el elemento 7 está en el conjunto) en su A $\langle\!\rangle_1$ .

Al igual que los clústeres, cada resumen es solo un conjunto dinámico con tamaño de universo pu, por lo que representamos cada resumen como un proto-EB. pu/ estructura. Los cuatro bits de resumen para la estructura proto-EB.16/ principal están en la estructura proto-EB.4/ más a la izquierda , y finalmente aparecen en dos estructuras proto-EB.2/ . Por ejemplo, la estructura proto-EB.2/ etiquetada como "grupos 2,3" tiene A $\langle\!\rangle_0$  D 0, lo que indica que el grupo 2 de la estructura proto-EB.16/ (que contiene los elementos 8; 9; 10; 11) es todo 0 y A $\langle\!\rangle_1$  D 1, lo que nos dice que el grupo 3 (que contiene los elementos 12; 13; 14; 15) tiene al menos un 1. Cada estructura proto-EB.4/ apunta a su propio resumen, que a su vez se almacena como un proto -EB.2/ estructura. Por ejemplo, mire la estructura proto-EB.2/ justo a la izquierda de la etiquetada como "elementos 0,1". Como su A $\langle\!\rangle_0$  es 0, nos dice que la estructura "elementos 0,1" es todo 0, y como su A $\langle\!\rangle_1$  es 1, sabemos que la estructura "elementos 2,3" contiene al menos un 1.

## 20.2.2 Operaciones sobre una estructura proto van Emde Boas

Ahora describiremos cómo realizar operaciones en una estructura proto-vEB.

Primero examinamos las operaciones de consulta (MIEMBRO, MÍNIMO, MÁXIMO y SUCESOR) que no cambian la estructura proto-vEB. Luego discutimos INSERTAR y ELIMINAR. Dejamos MÁXIMO y PREDECESOR, que son simétricos a MÍNIMO y SUCESOR, respectivamente, como Ejercicio 20.2-1.

Cada una de las opciones MIEMBRO, SUCESOR, PREDECESOR, INSERTAR y ELIMINAR toma un parámetro x, junto con una estructura proto-vEB V. Cada uno de estos operaciones supone que  $0 \leq x < V.u$ .

Determinar si un valor está en el conjunto

Para realizar MEMBER.x/, necesitamos encontrar el bit correspondiente a x dentro de la estructura proto-EB.2/ apropiada . Podemos hacerlo en O.lg lg u/ time, sin pasar por

las estructuras de resumen por completo. El siguiente procedimiento toma una estructura proto-EB V y un valor x, y devuelve un bit que indica si x está en el conjunto dinámico que tiene V.

```
PROTO-VEB-MIEMBRO.V; x/ 1
si V:u == 2
    devuelve
V:A&Ex 2 3 de lo contrario devuelve PROTO-VEB-MEMBER.V:cluster&high.x/; bajo.x//
```

El procedimiento PROTO-VEB-MIEMBRO funciona de la siguiente manera. La línea 1 prueba si estamos en un caso base, donde V es una estructura proto-EB.2/. La línea 2 maneja el caso base, simplemente devolviendo el bit apropiado de la matriz A. La línea 3 trata el caso recursivo, "profundizando" en la estructura proto-vEB más pequeña apropiada. El valor high.x/ dice qué proto-EB. pu/ estructura que visitamos, y low.x/ determina qué elemento dentro de ese proto-EB. pu/ estructura que estamos consultando.

Veamos qué pasa cuando llamamos a PROTO-VEB-MEMBER.V; 6/ sobre la estructura proto-EB.16/ de la figura 20.4. Desde high.6/ D 1 cuando u D 16, recurrimos a la estructura proto-EB.4/ en la parte superior derecha, y preguntamos por el elemento low.6/ D 2 de esa estructura. En esta llamada recursiva, u D 4, y así recurrimos de nuevo. Con u D 4, tenemos high.2/ D 1 y low.2/ D 0, por lo que preguntamos sobre el elemento 0 de la estructura proto-EB.2/ en la parte superior derecha. Esta llamada recursiva resulta ser un caso base, por lo que devuelve A&E0 D 0 a través de la cadena de llamadas recursivas. Por lo tanto, obtenemos el resultado de que PROTO-VEB-MEMBER.V; 6/ devuelve 0, lo que indica que 6 no está en el conjunto.

Para determinar el tiempo de ejecución de PROTO-VEB-MEMBER, denote con T .u/ su tiempo de ejecución en una estructura proto-EB.u/. Cada llamada recursiva toma un tiempo constante, sin incluir el tiempo que toman las llamadas recursivas que realiza. Cuando PROTO-VEB-MEMBER realiza una llamada recursiva, realiza una llamada en un proto-EB. pu/ estructura. Así, podemos caracterizar el tiempo de ejecución por la recurrencia T .u/ DT .pu/ C O.1/, que ya hemos visto como recurrencia (20.2). Su solución es T .u/ D O.lg lg u/, por lo que concluimos que PROTO-VEB-MEMBER corre en el tiempo O.lg lg u/.

### Encontrar el elemento mínimo

Ahora examinamos cómo realizar la operación MÍNIMA . El procedimiento PROTO-VEB-MINIMUM.V / devuelve el elemento mínimo en la estructura proto-vEB tura V , o NIL si V representa un conjunto vacío.

```

PROTO-VEB-MINIMUM.V /
1 si V:u == 2 2 si
V:ACE0 == 1 3 devuelve 0 4
de lo contrario V:ACE1 ==
1 5 devuelve 1 6 de lo contrario
devuelve NIL 7 de lo
contrario min-cluster D PROTO-
VEB-MINIMUM.V:summary/ 8
si min-cluster == NIL
9         devuelve NIL
10        else compensar D PROTO-VEB-MINIMUM.V:cluster<min-cluster/
11        volver index.min-cluster; compensar/

```

Este procedimiento funciona de la siguiente manera. La línea 1 prueba el caso base, que las líneas 2 a 6 manejan por fuerza bruta. Las líneas 7 a 11 manejan el caso recursivo. Primero, la línea 7 encuentra el número del primer grupo que contiene un elemento del conjunto. Lo hace llamando recursivamente a PROTO-VEB-MINIMUM en V:summary, que es un proto-EB. pu/ estructura. La línea 7 asigna este número de grupo a la variable min-cluster. Si el conjunto está vacío, la llamada recursiva devuelve NIL y la línea 9 devuelve NIL. De lo contrario, el elemento mínimo del conjunto se encuentra en algún lugar del número de clúster min-cluster. El la llamada recursiva en la línea 10 encuentra el desplazamiento dentro del grupo del elemento mínimo en este cúmulo. Finalmente, la línea 11 construye el valor del elemento mínimo a partir del número de clúster y el desplazamiento, y devuelve este valor.

Aunque consultar la información de resumen nos permite encontrar rápidamente el grupo que contiene el elemento mínimo, porque este procedimiento realiza dos llamadas recursivas en proto-EB. pu/ estructuras, no se ejecuta en  $O(\lg \lg u)$  tiempo en el peor de los casos. Si  $T(u)$  denota el peor tiempo para PROTO-VEB-MINIMUM en una estructura proto-EB.  $u$ , tenemos la recurrencia

$$T(u) = 2T(\frac{u}{2}) + O(1) \quad (20.3)$$

Nuevamente, usamos un cambio de variables para resolver esta recurrencia, dejando  $m = \lg u$ , lo que da  $T(2^m) = 2T(2^{m-1}) + O(1)$

$$2^m = 2T(2^{m-1}) + O(1)$$

Renombrando  $S_m = DT(2^m)$  da

$$S_m = 2S_{m-1} + O(1)$$

que, por el caso 1 del método maestro, tiene como solución  $S_m = O(m)$ . Al volver a cambiar de  $S_m$  a  $T(u)$ , tenemos que  $T(u) = DT(\lg u)$ . Por lo tanto, vemos que debido a la segunda llamada recursiva, PROTO-VEB MINIMUM se ejecuta en el tiempo  $\lg \lg u$  en lugar del tiempo  $\lg u$  deseado.

### Encontrar al sucesor

La operación SUCCESSOR es aún peor. En el peor de los casos, realiza dos llamadas recursivas, junto con una llamada a PROTO-VEB-MINIMUM. El procedimiento PROTO-VEB SUCCESSOR.V; x/ devuelve el elemento más pequeño en la estructura proto-vEB V que es mayor que x, o NIL si ningún elemento en V es mayor que x. No requiere que x sea un miembro del conjunto, pero asume que 0 x < V:u.

```

PROTO-VEB-SUCESOR.V; x/ 1 si
V:u == 2 2 si x
== 0 y V:A&1 == 1 3 devuelve 1 4
de lo contrario devuelve
NIL 5 de lo contrario
compensa D PROTO-VEB-SUCCESSOR.V:cluster&high.x/; low.x// 6 if offset ≠
NIL 7 return index.high.x/;
offset/ 8 else succ-cluster D PROTO-VEB-
SUCCESSOR.V:summary; alta.x// 9 10
si succ-cluster == NIL
devolver NIL
11 else compensar D PROTO-VEB-MINIMUM.V:cluster&succ-cluster/
12 volver index.succ-cluster; compensar/

```

El procedimiento PROTO-VEB-SUCCESSOR funciona de la siguiente manera. Como de costumbre, la línea 1 prueba el caso base, que las líneas 2 a 4 manejan por fuerza bruta: la única forma en que x puede tener un sucesor dentro de una estructura proto-EB.2/ es cuando x D 0 y A&1 es 1. Líneas 5–12 manejan el caso recursivo. La línea 5 busca un sucesor de x dentro del grupo de x, asignando el resultado a la compensación. La línea 6 determina si x tiene un sucesor dentro de su grupo; si es así, la línea 7 calcula y devuelve el valor de este sucesor. De lo contrario, tenemos que buscar en otros clústeres. La línea 8 asigna a succ-cluster el número del siguiente clúster no vacío, utilizando la información de resumen para encontrarlo. La línea 9 prueba si succ-cluster es NIL, y la línea 10 devuelve NIL si todos los clústeres subsiguientes están vacíos. Si succ-cluster no es NIL, la línea 11 asigna el primer elemento dentro de ese grupo para compensar, y la línea 12 calcula y devuelve el elemento mínimo en ese grupo.

En el peor de los casos, PROTO-VEB-SUCCESSOR se llama a sí mismo recursivamente dos veces en proto-EB. pu/ estructuras, y hace una llamada a PROTO-VEB-MINIMUM en un proto-EB. pu/ estructura. Por lo tanto, la recurrencia para el tiempo de ejecución del peor de los casos T .u/ de PROTO-VEB-SUCCESSOR es

$$T .u/ \leq T .pu/ C ,lg pu/ \leq T .pu/ C$$

—

,lg u/ :

Podemos emplear la misma técnica que usamos para la recurrencia (20.1) para mostrar que esta recurrencia tiene la solución  $T \leq D \cdot \lg u \lg \lg u$ . Por lo tanto, PROTO-VEB SUCCESSOR es asintóticamente más lento que PROTO-VEB-MINIMUM.

#### Insertar un elemento

Para insertar un elemento, debemos insertarlo en el grupo apropiado y también establecer el bit de resumen para ese grupo en 1. El procedimiento PROTO-VEB-INSERT.V; x/ inserta el valor x en la estructura proto-veb V .

```
PROTO-VEB-INSERT.V; x/ 1 if V:u
= = 2 2 V:A&Ex D
1 3 else PROTO-VEB-
INSERT.V:clusterŒhigh.x/; low.x// 4 PROTO-VEB-INSERT.V:resumen;
alto.x//
```

En el caso base, la línea 2 establece el bit apropiado en la matriz A en 1. En el caso recursivo, la llamada recursiva en la línea 3 inserta x en el grupo apropiado, y la línea 4 establece el bit de resumen para ese grupo en 1.

Debido a que PROTO-VEB-INSERT realiza dos llamadas recursivas en el peor de los casos, la recurrencia (20.3) caracteriza su tiempo de ejecución. Por lo tanto, PROTO-VEB-INSERT se ejecuta en  $\lg u$  tiempo.

#### Eliminación de un elemento

La operación DELETE es más complicada que la inserción. Mientras que siempre podemos establecer un bit de resumen en 1 al insertar, no siempre podemos restablecer el mismo bit de resumen en 0 al eliminar. Necesitamos determinar si algún bit en el grupo apropiado es 1. Como hemos definido estructuras proto-veb, tendríamos que examinar todos los bits pu dentro de un grupo para determinar si alguno de ellos es 1. Alternativamente, podríamos agregar un atributo n a la estructura proto-veb, contando cuántos elementos tiene. Dejamos la implementación de PROTO-VEB-DELETE como Ejercicios 20.2-2 y 20.2-3.

Claramente, necesitamos modificar la estructura proto-veb para que cada operación se reduzca a realizar como máximo una llamada recursiva. Veremos en el siguiente apartado cómo hacerlo.

#### Ejercicios

##### 20.2-1

Escriba pseudocódigo para los procedimientos PROTO-VEB-MAXIMO y PROTO-VEB PREDECESOR.

## 20.2-2

Escribir pseudocódigo para PROTO-VEB-DELETE. Debería actualizar el bit de resumen apropiado escaneando los bits relacionados dentro del clúster. ¿Cuál es el peor tiempo de ejecución de su procedimiento?

## 20.2-3

Agregue el atributo  $n$  a cada estructura proto-vEB, dando el número de elementos actualmente en el conjunto que representa, y escriba el pseudocódigo para PROTO-VEB-DELETE que usa el atributo  $n$  para decidir cuándo restablecer los bits de resumen a 0. ¿Cuál es el peor tiempo de ejecución de su procedimiento? ¿Qué otros procedimientos deben cambiar debido al nuevo atributo? ¿Estos cambios afectan sus tiempos de ejecución?

## 20.2-4

Modifique la estructura proto-vEB para admitir claves duplicadas.

## 20.2-5

Modifique la estructura proto-vEB para admitir claves que tengan datos satelitales asociados.

## 20.2-6

Escriba un pseudocódigo para un procedimiento que crea una estructura proto-EB.u/ .

## 20.2-7

Argumente que si se ejecuta la línea 9 de PROTO-VEB-MINIMUM , entonces la estructura proto-vEB está vacía.

## 20.2-8

Suponga que diseñamos una estructura proto-vEB en la que cada arreglo de conglomerados tenía solo  $u^{1/4}$  elementos. ¿Cuáles serían los tiempos de ejecución de cada operación?

---

## 20.3 El árbol de van Emde Boas

La estructura proto-vEB de la sección anterior está cerca de lo que necesitamos para lograr tiempos de ejecución  $O(\lg \lg u)$ . Se queda corto porque tenemos que recurrir demasiadas veces en la mayoría de las operaciones. En esta sección, diseñaremos una estructura de datos que es similar a la estructura proto-vEB pero almacena un poco más de información, eliminando así la necesidad de recursión.

En la sección 20.2, observamos que la suposición que hicimos sobre el tamaño del universo (que  $u \geq 2^{2k}$  para algún entero  $k$ ) es indebidamente restrictiva y limita los posibles valores de  $u$  a un conjunto excesivamente disperso. De aquí en adelante, por lo tanto, permitiremos que el tamaño del universo  $u$  sea cualquier potencia exacta de 2, y cuando  $u$  no sea un entero

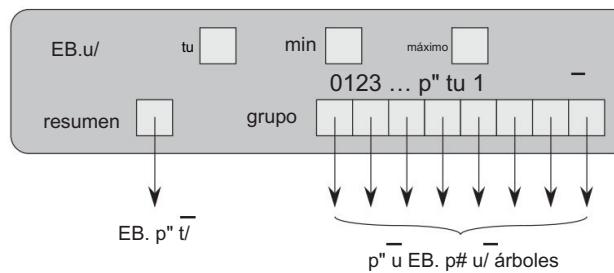


Figura 20.5 La información en un árbol EB.u/ cuando  $u > 2$ . La estructura contiene el tamaño del universo  $u$ , los elementos mínimo y máximo, un resumen de puntero a una EB.  $p"$  u/ árbol, y un arreglo cluster $\ominus$  $E0 :: p"$  u 1 de  $p"$  u punteros a EB.  $p\# u/ árboles$ .

ger, es decir, si  $u$  es una potencia impar de 2 ( $u \leq 2^k C_1$  para algún entero  $k \geq 0$ ), entonces dividiremos los  $\lg u$  bits de un número en los bits  $d. \lg u / 2e$  más significativos y los menos significativos  $b. \lg u / 2c$  bits. Por conveniencia, denotamos  $2d. \lg u / 2e$  (la “raíz cuadrada superior” de  $u$ ) por  $\underline{p}$  u y  $2b. \lg u / 2c$  (la “raíz cuadrada inferior” de  $u$ ) por  $\underline{p\#} u$ , entonces que  $\underline{u} = D \underline{p}$  u  $\underline{p\#} u$  y, cuando  $u$  es una potencia par de 2 ( $u \leq 2^k$  para algún entero  $k$ ),  $\underline{p}$  u  $D \underline{p\#} u$   $\underline{D} p$  u. Porque ahora permitimos que  $u$  sea una potencia impar de 2, debemos redefinir nuestras funciones útiles de la Sección 20.2:

```

alto.x/ D x=p# u ; bajo.x/ D
x mod p# u índice.x; y/ D x :
p# tu C y :

```

### 20.3.1 árboles van Emde Boas

El árbol de van Emde Boas, o árbol vEB, modifica la estructura proto-vEB. Denotamos un árbol vEB con un tamaño de universo de  $u$  como EB.u/ y, a menos que  $u$  sea igual al tamaño base de 2, el resumen de atributos apunta a un EB.  $p"$  u/ y el arreglo cluster $\ominus$  $E0 :: p"$  u 1 apunta a  $p"$  u EB.  $p\# u/ árboles$ . Como ilustra la figura 20.5, un árbol vEB contiene dos atributos que no se encuentran en una estructura proto-vEB:

min almacena el elemento mínimo en el árbol vEB, y  
 max almacena el elemento máximo en el árbol vEB.

Además, el elemento almacenado en min no aparece en ninguna de las EB recursivas  $. p\# u/ árboles$  a los que apunta la matriz de conglomerados . Los elementos almacenados en un árbol EB.u/ V , por lo tanto, son V:min más todos los elementos almacenados recursivamente en el EB.  $p\# u/ árboles$  señalados por V:cluster $\ominus$  $E0 :: p"$  u 1. Tenga en cuenta que cuando un árbol vEB contiene dos o más elementos, tratamos min y max de manera diferente: el elemento

almacenado en min no aparece en ninguno de los grupos, pero el elemento almacenado en max sí.

Dado que el tamaño base es 2, un árbol EB.2/ no necesita el arreglo A que tiene la estructura proto-EB.2/ correspondiente . En cambio, podemos determinar sus elementos a partir de sus atributos mínimo y máximo . En un árbol vEB sin elementos, independientemente del tamaño de su universo u, tanto el mínimo como el máximo son NIL.

La figura 20.6 muestra un árbol V EB.16/ que contiene el conjunto f2; 3; 4; 5; 7; 14; 15 g. Como el elemento más pequeño es 2, V:min es igual a 2, y aunque alto.2/D 0, el elemento 2 no aparece en el árbol EB.4/ al que apunta V:cluster $\ominus$ E0: observe que V:cluster $\ominus$ E0: min es igual a 3, por lo que 2 no está en este árbol vEB. De manera similar, dado que V:cluster $\ominus$ E0:min es igual a 3, y 2 y 3 son los únicos elementos en V:cluster $\ominus$ E0, los clústeres EB.2/ dentro de V:cluster $\ominus$ E0 están vacíos.

Los atributos min y max serán clave para reducir el número de llamadas recursivas dentro de las operaciones en los árboles vEB. Estos atributos nos ayudarán de cuatro formas:

1. Las operaciones MÍNIMO y MÁXIMO ni siquiera necesitan repetirse, ya que solo puede devolver los valores de min o max.
2. La operación SUCCESSOR puede evitar realizar una llamada recursiva para determinar si el sucesor de un valor x se encuentra dentro de high.x/. Esto se debe a que el sucesor de x se encuentra dentro de su grupo si y solo si x es estrictamente menor que el atributo máximo de su grupo. Un argumento simétrico vale para PREDECESOR y min.
3. Podemos saber si un árbol vEB no tiene elementos, exactamente un elemento o al menos dos elementos en tiempo constante a partir de sus valores mínimo y máximo . Esta habilidad ayudará en las operaciones INSERTAR y ELIMINAR . Si min y max son ambos NIL, entonces el árbol vEB no tiene elementos. Si min y max no son NIL pero son iguales entre sí, entonces el árbol vEB tiene exactamente un elemento. De lo contrario, tanto min como max no son NIL pero son desiguales, y el árbol vEB tiene dos o más elementos.
4. Si sabemos que un árbol vEB está vacío, podemos insertar un elemento en él actualizando solo sus atributos mínimo y máximo . Por lo tanto, podemos insertar en un árbol vEB vacío en tiempo constante. De manera similar, si sabemos que un árbol vEB tiene solo un elemento, podemos eliminar ese elemento en tiempo constante actualizando solo min y max. Estas propiedades nos permitirán acortar la cadena de llamadas recursivas.

Incluso si el tamaño del universo u es una potencia impar de 2, la diferencia en los tamaños del árbol vEB de resumen y los clústeres no afectará los tiempos de ejecución asintóticos de las operaciones del árbol vEB. Todos los procedimientos recursivos que implementan las operaciones del árbol vEB tendrán tiempos de ejecución caracterizados por la recurrencia

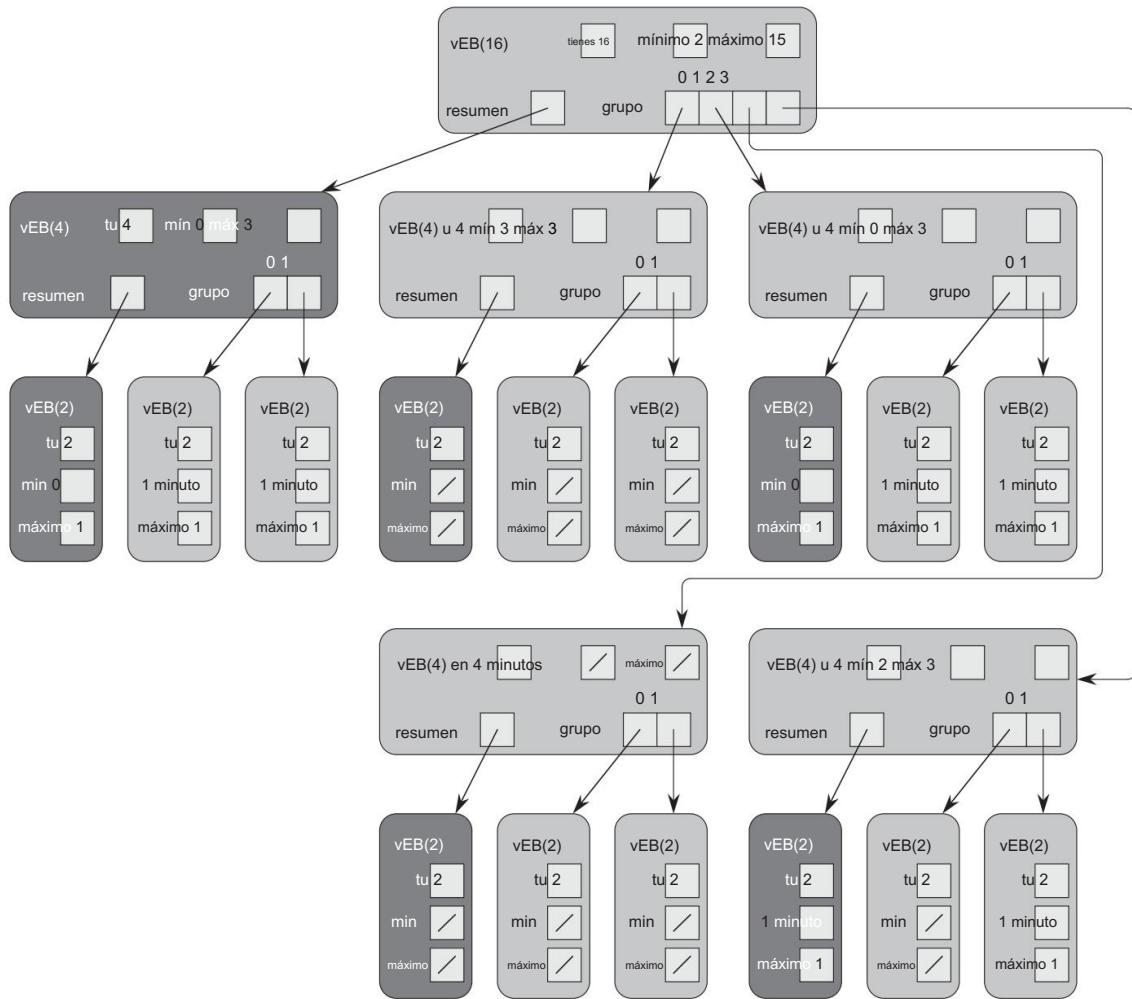


Figura 20.6 Un árbol EB.16/ correspondiente al árbol proto-vEB de la Figura 20.4. Almacena el conjunto f2; 3; 4; 5; 7; 14; 15 g. Las barras indican valores NIL . El valor almacenado en el atributo min de un árbol vEB no aparece en ninguno de sus clústeres. El sombreado intenso tiene el mismo propósito aquí que en la figura 20.4.

Esta recurrencia se parece a la recurrencia (20.2) y la resolveremos de manera similar. Haciendo  $m = D \lg u$ , lo reescribimos como  $T .2m/ T .2dm=2e / C$

O.1/ : Observando que  $dm=2e$

$2m=3$  para todo  $m \geq 2$ , tenemos  $T .2m/ T .22m=3 / C$  O.1/ :

Haciendo  $Sm/ DT .2m/$ , reescribimos

esta última recurrencia como

$Sm/ S.2m=3/ C$  O.1/ ;

que, por el caso 2 del método maestro, tiene como solución  $Sm/ D O.\lg m/$ . (En términos de la solución asintótica, la fracción  $2=3$  no hace ninguna diferencia en comparación con la fracción  $1=2$ , porque cuando aplicamos el método maestro, encontramos que  $\log_3=2 \geq 1$  D  $\log_2 1 \geq 1$  D 0:) Por lo tanto, tenemos  $T .u/ DT .2m/ D Sm/ D O.\lg m/ D O.\lg \lg u/$ .

Antes de usar un árbol de van Emde Boas, debemos conocer el tamaño del universo  $u$ , de modo que podamos crear un árbol de van Emde Boas del tamaño apropiado que inicialmente represente un conjunto vacío. Como el problema 20-1 le pide que muestre, el requisito de espacio total de un árbol de van Emde Boas es  $Ou/$ , y es sencillo crear un árbol vacío en  $Ou/$  tiempo. Por el contrario, podemos crear un árbol rojo-negro vacío en tiempo constante.

Por lo tanto, es posible que no queramos usar un árbol de van Emde Boas cuando realizamos solo una pequeña cantidad de operaciones, ya que el tiempo para crear la estructura de datos excedería el tiempo ahorrado en las operaciones individuales. Este inconveniente generalmente no es significativo, ya que normalmente usamos una estructura de datos simple, como una matriz o una lista enlazada, para representar un conjunto con solo unos pocos elementos.

### 20.3.2 Operaciones en un árbol de van Emde Boas

Ahora estamos listos para ver cómo realizar operaciones en un árbol de Van Emde Boas. Como hicimos con la estructura proto van Emde Boas, consideraremos primero las operaciones de consulta y luego INSERT y DELETE. Debido a la ligera asimetría entre los elementos mínimo y máximo en un árbol vEB (cuando un árbol vEB contiene al menos dos elementos, el elemento mínimo no aparece dentro de un grupo pero el elemento máximo sí) proporcionaremos un pseudocódigo para las cinco operaciones de consulta. Es igual que en las operaciones sobre protoestructuras de van Emde Boas, las operaciones aquí que toman los parámetros  $V$  y  $x$ , donde  $V$  es un árbol de van Emde Boas y  $x$  es un elemento, asumen que  $0 \leq x < V.u$ .

#### Encontrar los elementos mínimo y máximo

Debido a que almacenamos el mínimo y el máximo en los atributos min y max, dos de las operaciones son de una sola línea y toman un tiempo constante:

VEB-ÁRBOL-MINIMO.V / 1

vuelta V:min

VEB-ÁRBOL-MAXIMO.V /

1 vuelta V:máx

Determinar si un valor está en el conjunto

El procedimiento VEB-ÁRBOL-MIEMBRO.V; x/ tiene un caso recursivo como el de PROTO-VEB-MEMBER, pero el caso base es un poco diferente. También comprobamos directamente si x es igual al elemento mínimo o máximo. Dado que un árbol vEB no almacena bits como lo hace una estructura proto-vEB, diseñamos VEB-TREE-MEMBER para que devuelva VERDADERO o FALSO en lugar de 1 o 0.

VEB-ÁRBOL-MIEMBRO.V; x/ 1

si x == V:mín o x == V:máx 2

devolver VERDADERO

3 más si V:u == 2

4 devuelve FALSO

5 else devuelve VEB-TREE-MEMBER.V:clusterŒhigh.x/; bajo.x//

La línea 1 verifica si x es igual al elemento mínimo o máximo.

Si es así, la línea 2 devuelve VERDADERO. De lo contrario, la línea 3 prueba el caso base.

Dado que un árbol EB.2/ no tiene más elementos que los de min y max, si es el caso base, la línea 4 devuelve FALSO. La otra posibilidad, no es un caso base y x no es igual a min ni max, es manejada por la llamada recursiva en la línea 5.

La recursión (20.4) caracteriza el tiempo de ejecución del procedimiento VEB-ÁRBOL-MIEMBRO , por lo que este procedimiento toma O.lg lg u/ tiempo.

Encontrar el sucesor y el predecesor

A continuación vemos cómo implementar la operación SUCCESSOR . Recuérdese que el procedimiento PROTO-VEB-SUCCESSOR.V; x/ podría hacer dos llamadas recursivas: una para determinar si el sucesor de x reside en el mismo grupo que x y, si no es así, otra para encontrar el grupo que contiene al sucesor de x. Debido a que podemos acceder rápidamente al valor máximo en un árbol vEB, podemos evitar realizar dos llamadas recursivas y, en su lugar, realizar una llamada recursiva en un clúster o en el resumen, pero no en ambos.

```

VEB-ÁRBOL-SUCESOR.V; x/ 1 si V:u
  == 2 2 si x == 0
  y V:max == 1 3 devuelve 1 4 de lo
  contrario devuelve NIL 5
  de lo contrario V:min ≠ NIL y
  x < V:min 6 devuelve V:min 7 de lo contrario max-
  low D VEB-TREE-
MAXIMUM.V:clusterŒhigh.x// 8 si max-low ≠ NIL and low.x/ < max-low offset D
VEB-TREE-SUCCESSOR.V:clusterŒhigh.x/; bajo.x// devuelve
  9           índice.alto.x/; offset/ else succ-cluster D VEB-TREE-SUCCESSOR.V:summary;
  10          alto.x// 
  11
  12          si succ-cluster == NIL
  13          devuelve NIL
  14          más compensación D VEB-TREE-MINIMUM.V:clusterŒsucc-cluster/
  15          volver index.succ-cluster; compensar/

```

Este procedimiento tiene seis declaraciones de devolución y varios casos. Comenzamos con el caso base en las líneas 2 a 4, que devuelve 1 en la línea 3 si estamos tratando de encontrar el sucesor de 0 y 1 está en el conjunto de 2 elementos; de lo contrario, el caso base devuelve NIL en la línea 4.

Si no estamos en el caso base, a continuación verificamos en la línea 5 si x es estrictamente menor que el elemento mínimo. Si es así, simplemente devolvemos el elemento mínimo en la línea 6.

Si llegamos a la línea 7, entonces sabemos que no estamos en un caso base y que x es mayor o igual que el valor mínimo en el árbol vEB V . La línea 7 asigna a max-low el elemento máximo en el grupo de x. Si el grupo de x contiene algún elemento que es mayor que x, entonces sabemos que el sucesor de x se encuentra en algún lugar dentro del grupo de x. La línea 8 prueba esta condición. Si el sucesor de x está dentro del grupo de x, entonces la línea 9 determina en qué parte del grupo está y la línea 10 devuelve el sucesor de la misma manera que la línea 7 de PROTO-VEB-SUCCESSOR.

Llegamos a la línea 11 si x es mayor o igual que el elemento más grande en su grupo. En este caso, las líneas 11 a 15 encuentran el sucesor de x de la misma manera que las líneas 8 a 12 de PROTO-VEB-SUCCESSOR.

Es fácil ver cómo la recursión (20.4) caracteriza el tiempo de ejecución de VEB TREE-SUCCESSOR. Dependiendo del resultado de la prueba en la línea 7, el procedimiento se llama a sí mismo recursivamente en la línea 9 (en un árbol vEB con tamaño de universo p# u) o en la línea 11 (en un árbol vEB con tamaño de universo p" u). caso, la única llamada recursiva está en un árbol vEB con un tamaño de universo como máximo p" u. El resto del procedimiento, incluidas las llamadas a VEB-ÁRBOL-MÍNIMO y VEB-ÁRBOL-MÁXIMO, dura 0,1/. Por lo tanto, VEB-TREE-SUCCESSOR se ejecuta en O.lg lg u/ en el peor de los casos.

El procedimiento VEB-ÁRBOL-PREDECESOR es simétrico al VEB-ÁRBOL procedimiento SUCCESSOR , pero con un caso adicional:

```

VEB-ÁRBOL-ANTECESOR.V; x/ 1 si
V:u == 2 si x ==
    1 y V:min == 0
        volver 0
    2 3 4 else devuelve NIL 5
elseif V:max ≠ NIL y x > V:max 6 return
V:max 7 else min-low D
VEB-TREE-MINIMUM.V:clusterŒhigh.x// 8 if min-low ≠ NIL y bajo.x/ >
min-bajo 9 desplazamiento D VEB-ÁRBOL-
PREDECESOR.V:clusterŒalto.x/; bajo.x// 10 índice de retorno.alto.x/; offset/ 11 else
pred-cluster D VEB-TREE-
PREDECESSOR.V:summary; alto.x// 12 13
    si pred-cluster == NIL si
        V:min ≠ NIL y x > V:min devuelve
14            V:min
15            de lo contrario, devuelve NIL
else compensa D VEB-TREE-MAXIMUM.V:clusterŒpred-cluster/
17            volver index.pred-cluster; compensar/

```

Las líneas 13 y 14 forman el caso adicional. Este caso ocurre cuando el predecesor de x, si existe, no reside en el clúster de x. En VEB-TREE-SUCCESSOR, se nos aseguró que si el sucesor de x reside fuera del grupo de x, entonces debe residir en un grupo con un número más alto. Pero si el predecesor de x es el valor mínimo en el árbol vEB V, entonces el sucesor no reside en ningún clúster. La línea 13 comprueba esta condición y la línea 14 devuelve el valor mínimo según corresponda.

Este caso adicional no afecta el tiempo de ejecución asintótico de VEB-TREE PREDECESOR en comparación con VEB-TREE-SUCCESSOR, por lo que VEB TREE-PREDECESOR se ejecuta en O.lg lg u/ en el peor de los casos.

### Insertar un elemento

Ahora examinamos cómo insertar un elemento en un árbol vEB. Recuerde que PROTO VEB-INSERT hizo dos llamadas recursivas: una para insertar el elemento y otra para insertar el número de grupo del elemento en el resumen. El procedimiento VEB-TREE-INSERT hará solo una llamada recursiva. ¿Cómo podemos salirnos con la nuestra? Cuando insertamos un elemento, el grupo al que pertenece ya tiene otro elemento o no. Si el clúster ya tiene otro elemento, entonces el número de clúster ya está en el resumen, por lo que no necesitamos hacer esa llamada recursiva. Si

el clúster aún no tiene otro elemento, entonces el elemento que se inserta se convierte en el único elemento en el clúster, y no necesitamos repetir para insertar un elemento en un árbol vEB vacío:

VEB-ÁRBOL-VACÍO-INSERTAR.V; x/ 1

V:mín D x 2 V:máx

D x

Con este procedimiento en mano, aquí está el pseudocódigo para VEB-TREE-INSERT.V; x/, que asume que x no es ya un elemento en el conjunto representado por vEB árbol V :

VEB-ÁRBOL-INSERTO.V; X/

1 si V:min == NIL

    VEB-ÁRBOL-VACÍO-INSERTAR.V; x/ 2 3

    si x < V:min

        4           intercambiar x con V:min si

        5           V:u > 2 si

        6           VEB-TREE-MINIMUM.V:clusterŒhigh.x// == NIL VEB-TREE-

        7           INSERT.V:summary; high.x// VEB-EMPTY-TREE-

        8           INSERT.V:clusterŒhigh.x/; bajo.x//

        9           else VEB-ÁRBOL-INSERT.V:clusterŒhigh.x/; bajo.x//

        10          si x > V:max

        11          V: máx D x

Este procedimiento funciona de la siguiente manera. La línea 1 prueba si V es un árbol vEB vacío y, si lo es, la línea 2 maneja este caso sencillo. Las líneas 3 a 11 suponen que V no está vacío y, por lo tanto, algún elemento se insertará en uno de los grupos de V. Pero ese elemento podría no ser necesariamente el elemento x pasado a VEB-TREE-INSERT.

Si x < min, como se probó en la línea 3, entonces x debe convertirse en el nuevo min. Sin embargo, no queremos perder el min original , por lo que debemos insertarlo en uno de los grupos de V. En este caso, la línea 4 intercambia x con min, de modo que insertamos el min original en uno de los grupos de V.

Ejecutamos las líneas 6 a 9 solo si V no es un árbol vEB de caso base. La línea 6 determina si el grupo al que irá x está actualmente vacío. Si es así, entonces la línea 7 inserta el número de grupo de x en el resumen y la línea 8 maneja el caso sencillo de insertar x en un grupo vacío. Si el grupo de x no está actualmente vacío, entonces la línea 9 inserta x en su grupo. En este caso, no necesitamos actualizar el resumen, ya que el número de clúster de x ya es miembro del resumen.

Finalmente, las líneas 10 y 11 se encargan de actualizar max si x > max. Tenga en cuenta que si V es un árbol vEB de caso base que no está vacío, las líneas 3–4 y 10–11 actualizan min y max correctamente.

Una vez más, podemos ver fácilmente cómo la recurrencia (20.4) caracteriza el tiempo de ejecución. Dependiendo del resultado de la prueba en la línea 6, ya sea la llamada recursiva en la línea 7 (ejecutada en un árbol vEB con tamaño de universo  $p^u$ ) o la llamada recursiva en la línea 9 (ejecutada en un vEB con tamaño de universo  $p^{\#u}$ ). En cualquier caso, la única llamada recursiva está en un árbol vEB con tamaño de universo como máximo  $p^u$ . Debido a que el resto de VEB TREE-INSERT toma  $O(1)$  tiempo, se aplica la recurrencia (20.4), por lo que el tiempo de ejecución es  $O(\lg \lg u)$ .

#### Eliminación de un elemento

Finalmente, veremos cómo eliminar un elemento de un árbol vEB. El procedimiento VEB-TREE-DELETE.V; x/ asume que x es actualmente un elemento en el conjunto representado por el árbol vEB V.

```

VEB-ÁRBOL-BORRAR.V; x/ 1
    si V:min == V:max 2 V:min
    D NIL V:max D NIL 4
    3      elseif V:u == 2

    5      si x == 0
    6          V:mín D 1
    7      más V:min D 0
    8      V:máx D V:mín 9 si
    no x == V:mín
    10     primer grupo D VEB-TREE-MINIMUM.V:summary/ x D
    11     index.first-cluster; VEB-
            TREE-MINIMUM.V:clusterŒfirst-cluster// V:min D x VEB-
    12     TREE-
    13     DELETE.V:clusterŒhigh.x/; low.x// if VEB-TREE-
    14     MINIMUM.V:clusterŒhigh.x// == NIL VEB-TREE-
    15     DELETE.V:resumen; alto.x// si x == V:max
    16
    17     resumen-max D VEB-TREE-MAXIMUM.V:resumen/ if resumen-
    18     max == NIL V:max D V:min
    19         else V:max D
    20     index.summary-max; VEB-TREE-
            MAXIMUM.V:clusterŒsummary-max// elseif x == V:max
    21
    22     V:máx D índice.alto.x/; VEB-
            ÁRBOL-MAXIMO.V:clusterŒhigh.x///

```

El procedimiento VEB-TREE-DELETE funciona de la siguiente manera. Si el árbol vEB V contiene solo un elemento, entonces es tan fácil eliminarlo como lo fue insertar un elemento en un árbol vEB vacío: simplemente establezca min y max en NIL. Las líneas 1 a 3 manejan este caso.

De lo contrario, V tiene al menos dos elementos. La línea 4 comprueba si V es un árbol vEB de caso base y, de ser así, las líneas 5 a 8 establecen el mínimo y el máximo para el único elemento restante.

Las líneas 9 a 22 asumen que V tiene dos o más elementos y que u 4. En este caso, tendremos que eliminar un elemento de un grupo. Sin embargo, es posible que el elemento que eliminemos de un clúster no sea x, porque si x es igual a min, una vez que hayamos eliminado x, algún otro elemento dentro de uno de los clústeres de V se convierte en el nuevo min, y tenemos que eliminar ese otro elemento. de su racimo. Si la prueba en la línea 9 revela que estamos en este caso, entonces la línea 10 establece first-cluster en el número del grupo que contiene el elemento más bajo que no sea min, y la línea 11 establece x en el valor del elemento más bajo en ese grupo. Este elemento se convierte en el nuevo min en la línea 12 y, debido a que establecemos x en su valor, es el elemento que se eliminará de su grupo.

Cuando llegamos a la línea 13, sabemos que necesitamos eliminar el elemento x de su grupo, ya sea x el valor pasado originalmente a VEB-TREE-DELETE o si x es el elemento que se convierte en el nuevo mínimo. La línea 13 elimina x de su grupo.

Ese grupo ahora podría quedar vacío, lo que prueba la línea 14, y si lo hace, entonces debemos eliminar el número de grupo de x del resumen, que maneja la línea 15.

Después de actualizar el resumen, es posible que debamos actualizar max. La línea 16 verifica si estamos eliminando el elemento máximo en V y, si lo estamos, entonces la línea 17 establece summary-max en el número del grupo no vacío con el número más alto. (La llamada VEB-TREE-MAXIMUM.V:summary/ funciona porque ya hemos llamado recursivamente a VEB-TREE-DELETE en V:summary y, por lo tanto, V:summary:max ya se ha actualizado según sea necesario). Si todo V Los grupos de están vacíos, entonces el único elemento restante en V es min; la línea 18 verifica este caso y la línea 19 actualiza el máximo de manera adecuada. De lo contrario, la línea 20 establece max en el elemento máximo en el grupo con el número más alto. (Si este grupo es donde se eliminó el elemento, nuevamente confiamos en que la llamada recursiva en la línea 13 ya corrigió el atributo máximo de ese grupo ).

Finalmente, tenemos que manejar el caso en el que el clúster de x no se vació debido a que se eliminó x. Aunque no tenemos que actualizar el resumen en este caso, es posible que tengamos que actualizar max. La línea 21 prueba para este caso, y si tenemos que actualizar max, la línea 22 lo hace (de nuevo confiando en la llamada recursiva para haber corregido max en el clúster).

Ahora mostramos que VEB-TREE-DELETE se ejecuta en O.lg lg u/ time en el peor de los casos. A primera vista, podría pensar que la recurrencia (20.4) no siempre se aplica, porque una sola llamada de VEB-TREE-DELETE puede hacer dos llamadas recursivas: una en la línea 13 y otra en la línea 15. Aunque el procedimiento puede hacer ambas llamadas recursivas llamadas, vamos a pensar en lo que sucede cuando lo hace. Para que la llamada recursiva en

Para que ocurra la línea 15, la prueba en la línea 14 debe mostrar que el grupo de  $x$  está vacío. La única forma en que el grupo de  $x$  puede estar vacío es si  $x$  era el único elemento en su grupo cuando hicimos la llamada recursiva en la línea 13. Pero si  $x$  era el único elemento en su grupo, entonces esa llamada recursiva tomó  $O.1/\text{tiempo}$ , porque solo ejecutó las líneas 1–3. Así, tenemos dos posibilidades mutuamente excluyentes:

La llamada recursiva en la línea 13 tomó tiempo constante.

La llamada recursiva en la línea 15 no ocurrió.

En cualquier caso, la recurrencia (20.4) caracteriza el tiempo de ejecución de VEB-TREE DELETE y, por lo tanto, su tiempo de ejecución en el peor de los casos es  $O.\lg \lg u/$ .

### Ejercicios

20.3-1

Modificar árboles vEB para admitir claves duplicadas.

20.3-2

Modifique los árboles vEB para admitir claves que tengan datos satelitales asociados.

20.3-3

Escriba un pseudocódigo para un procedimiento que cree un árbol vacío de van Emde Boas.

20.3-4

¿Qué sucede si llama VEB-TREE-INSERT con un elemento que ya está en el árbol vEB? ¿Qué sucede si llama a VEB-TREE-DELETE con un elemento que no está en el árbol vEB? Explique por qué los procedimientos muestran el comportamiento que muestran. Muestre cómo modificar árboles vEB y sus operaciones para que podamos verificar en tiempo constante si un elemento está presente.

20.3-5

Suponga que en lugar de  $p^u$  conglomerados, cada uno con tamaño de universo  $p\# u$ , construimos árboles vEB para tener  $u_1=k$  conglomerados, cada uno con tamaño de universo  $u_{11}=k$ , donde  $k>1$  es una constante. Si tuviéramos para modificar las operaciones apropiadamente, ¿cuáles serían sus tiempos de ejecución? Para propósitos de análisis, suponga que  $u_1=k$  y  $u_{11}=k$  son siempre números enteros.

20.3-6

La creación de un árbol vEB con tamaño de universo  $u$  requiere  $O_u/\text{tiempo}$ . Supongamos que deseamos dar cuenta explícitamente de ese tiempo. ¿Cuál es el menor número de operaciones  $n$  para el cual el tiempo amortizado de cada operación en un árbol vEB es  $O.\lg \lg u/$ ?

## Problemas

### 20-1 Requerimientos de espacio para los árboles de van Emde Boas

Este problema explora los requerimientos de espacio para los árboles de van Emde Boas y sugiere una forma de modificar la estructura de datos para hacer que su requerimiento de espacio dependa del número  $n$  de elementos realmente almacenados en el árbol, en lugar de que en el tamaño del universo  $u$ . Para simplificar, suponga que  $pu$  es siempre un número

a. Explique por qué la siguiente recurrencia caracteriza el requisito de espacio  $P_u$  de un árbol de van Emde Boas con tamaño de universo  $u$ :

$$P_u = D_{pu} C \frac{1}{P_{pu}} + C_{pu} \quad (20.5)$$

b. Demuestre que la recurrencia (20.5) tiene la solución  $P_u = D \cdot O(u)$ .

Para reducir los requisitos de espacio, definamos un árbol van Emde Boas de espacio reducido, o árbol RS-vEB, como un árbol vEB V pero con los siguientes cambios:

El atributo  $V:cluster$ , en lugar de almacenarse como una simple matriz de punteros a árboles vEB con tamaño de universo  $pu$ , es una tabla hash (consulte el Capítulo 11) almacenada como una tabla dinámica (consulte la Sección 17.4). En correspondencia con la versión de matriz de  $V:cluster$ , la tabla hash almacena punteros a árboles RS-vEB con tamaño de universo  $pu$ . Para encontrar el  $i$ -ésimo grupo, buscamos la clave  $i$  en la tabla hash, de modo que podamos encontrar el  $i$ -ésimo grupo mediante una sola búsqueda en la tabla hash.

La tabla hash almacena solo punteros a clústeres no vacíos. Una búsqueda en la tabla hash de un clúster vacío devuelve NIL, lo que indica que el clúster está vacío.

El atributo  $V:summary$  es NIL si todos los clústeres están vacíos. De lo contrario,  $V:summary$  apunta a un árbol RS-vEB con tamaño de universo  $pu$ .

Debido a que la tabla hash se implementa con una tabla dinámica, el espacio que requiere es proporcional al número de clústeres no vacíos.

Cuando necesitamos insertar un elemento en un árbol RS-vEB vacío, creamos el árbol RS-vEB llamando al siguiente procedimiento, donde el parámetro  $u$  es el tamaño del universo del árbol RS-vEB:

**CREATE-NEW-RS-VEB-TREE.u** / 1 asigna

un nuevo árbol vEB V 2  $V:u$  D u 3

$V:min$  D NIL 4

$V:max$  D NIL 5

$V:summary$  D NIL 6

crea  $V:cluster$  como un tabla

hash dinámica vacía 7 devuelve V

- C. Modifique el procedimiento VEB-TREE-INSERT para producir un pseudocódigo para el procedimiento RS-VEB-TREE-INSERT.V; x/, que inserta x en el árbol RS-vEB V llamando CREATE-NEW-RS-VEB-TREE según corresponda.
- d. Modifique el procedimiento VEB-TREE-SUCCESSOR para producir un pseudocódigo para el procedimiento RS-VEB-TREE-SUCCESSOR.V; x/, que devuelve el sucesor de x en RS-vEB árbol V , o NIL si x no tiene sucesor en V .
- mi. Demuestre que, bajo el supuesto de hashing uniforme simple, sus procedimientos RS-VEB TREE-INSERT y RS-VEB-TREE-SUCCESSOR se ejecutan en O.lg lg u/ tiempo esperado.
- F. Suponiendo que los elementos nunca se eliminan de un árbol vEB, demuestre que el requisito de espacio para la estructura del árbol RS-vEB es On/, donde n es el número de elementos realmente almacenados en el árbol RS-vEB.
- gramo. Los árboles RS-vEB tienen otra ventaja sobre los árboles vEB: requieren menos tiempo para crearse. ¿Cuánto tiempo lleva crear un árbol RS-vEB vacío?

## 20-2 intentos y-rápidos

Este problema investiga los “intentos y-rápidos” de D. Willard que, como los árboles de van Emde Boas, realizan cada una de las operaciones MIEMBRO, MÍNIMO, MÁXIMO, PRE DECESOR y SUCESOR en elementos extraídos de un universo con tamaño u en O.lg lg u/ tiempo en el peor de los casos. Las operaciones INSERT y DELETE toman O.lg lg u/ tiempo amortizado. Al igual que los árboles de van Emde Boas de espacio reducido (vea el problema 20-1), los intentos rápidos usan solo el espacio On/ para almacenar n elementos. El diseño de los intentos y-fast se basa en un hashing perfecto (consulte la Sección 11.5).

Como estructura preliminar, suponga que creamos una tabla hash perfecta que contiene no solo todos los elementos del conjunto dinámico, sino todos los prefijos de la representación binaria de todos los elementos del conjunto. Por ejemplo, si u D 16, de modo que lg u D 4 y x D 13 están en el conjunto, entonces debido a que la representación binaria de 13 es 1101, la tabla hash perfecta contendría las cadenas 1, 11, 110 y 1101 Además de la tabla hash, creamos una lista doblemente enlazada de los elementos actualmente en el conjunto, en orden creciente.

- ¿Cuánto espacio requiere esta estructura?
- Muestre cómo realizar las operaciones MÍNIMO y MÁXIMO en tiempo O.1/; las operaciones MIEMBRO, PREDECESOR y SUCESOR en tiempo O.lg lg u/; y las operaciones INSERT y DELETE en tiempo O.lg u/.

Para reducir el requisito de espacio a On/, realizamos los siguientes cambios en el estructura de datos:

Agrupamos los  $n$  elementos en  $n = \lg u$  grupos de tamaño  $\lg u$ . (Supongamos por ahora que  $\lg u$  divide a  $n$ ). El primer grupo consta de los elementos  $\lg u$  más pequeños del conjunto, el segundo grupo consta de los siguientes elementos  $\lg u$  más pequeños, y así sucesivamente.

Designamos un valor “representativo” para cada grupo. El representante del  $i$ -ésimo grupo es al menos tan grande como el elemento más grande del  $i$ -ésimo grupo, y es más pequeño que todos los elementos del  $i$ -ésimo grupo. (El representante del último grupo puede ser el máximo elemento posible  $u - 1$ ). Tenga en cuenta que un representante puede ser un valor que no se encuentra actualmente en el conjunto.

Almacenamos los elementos  $\lg u$  de cada grupo en un árbol de búsqueda binaria equilibrado, como un árbol rojo-negro. Cada representante apunta al árbol de búsqueda binaria balanceada para su grupo, y cada árbol de búsqueda binaria balanceada apunta al representante de su grupo.

La tabla hash perfecta almacena solo los representantes, que también se almacenan en una lista doblemente enlazada en orden creciente.

Llamamos a esta estructura un trie y-rápido.

C. Muestre que un ensayo y-rápido requiere sólo espacio  $O(n)$  para almacenar  $n$  elementos.

d. Muestre cómo realizar las operaciones MÍNIMAS y MÁXIMAS en  $O(\lg \lg u)$  tiempo con un trie y-fast.

mi. Muestre cómo realizar la operación MEMBER en tiempo  $O(\lg \lg u)$ .

F. Muestre cómo realizar las operaciones PREDECESOR y SUCCESSOR en tiempo  $O(\lg \lg u)$ .

gramo. Explique por qué las operaciones INSERT y DELETE toman  $O(\lg \lg u)$  tiempo.

H. Muestre cómo relajar el requisito de que cada grupo en un intento y-rápido tenga exactamente  $\lg u$  elementos para permitir que INSERT y DELETE se ejecuten en  $O(\lg \lg u)$  tiempo amortizado sin afectar los tiempos asintóticos de ejecución de las otras operaciones.

## Notas del capítulo

La estructura de datos de este capítulo lleva el nombre de P. van Emde Boas, quien describió una forma temprana de la idea en 1975 [339]. Artículos posteriores de van Emde Boas [340] y van Emde Boas, Kaas y Zijlstra [341] refinaron la idea y la exposición.

Mehlhorn y Nüher [252] posteriormente ampliaron las ideas para aplicarlas al universo

tamaños que son primos. El libro de Mehlhorn [249] contiene un tratamiento ligeramente diferente de los árboles de van Emde Boas que el de este capítulo.

Usando las ideas detrás de los árboles de van Emde Boas, Dementiev et al. [83] desarrollaron un árbol de búsqueda no recursivo de tres niveles que corría más rápido que los árboles de van Emde Boas en sus propios experimentos.

Wang y Lin [347] diseñaron una versión canalizada por hardware de los árboles de van Emde Boas, que logra un tiempo amortizado constante por operación y utiliza  $O(\lg \lg u)$  etapas en la canalización.

Un límite inferior de  $P^{\text{lower}}_{\text{atras}, cu}$  y Thorup [273, 274] para encontrar el predecesor muestra que los árboles de van Emde Boas son óptimos para esta operación, incluso si se permite la aleatorización.

---

## 21

## Estructuras de datos para conjuntos disjuntos

Algunas aplicaciones implican agrupar  $n$  elementos distintos en una colección de conjuntos disjuntos. Estas aplicaciones a menudo necesitan realizar dos operaciones en particular: encontrar el conjunto único que contiene un elemento dado y unir dos conjuntos. Este capítulo explora métodos para mantener una estructura de datos que admita estas operaciones.

La Sección 21.1 describe las operaciones soportadas por una estructura de datos de conjuntos disjuntos y presenta una aplicación simple. En la Sección 21.2, observamos una implementación de lista enlazada simple para conjuntos disjuntos. La Sección 21.3 presenta una representación más eficiente usando árboles enraizados. El tiempo de ejecución usando la representación de árbol es teóricamente superlineal, pero para todos los propósitos prácticos es lineal. La sección 21.4 define y analiza una función de crecimiento muy rápido y su inversa de crecimiento muy lento, que aparece en el tiempo de ejecución de las operaciones en la implementación basada en árboles y luego, mediante un análisis amortizado complejo, demuestra un límite superior en el tiempo de ejecución que es apenas superlineal.

---

### 21.1 Operaciones de conjuntos disjuntos

Una estructura de datos de conjunto disjunto mantiene una colección  $S = S_1 \cup S_2 \cup \dots \cup S_k$  de conjuntos dinámicos disarticulares. Identificamos cada conjunto por un representante, que es algún miembro del conjunto. En algunas aplicaciones, no importa qué miembro se utilice como representante; solo nos importa que si preguntamos por el representante de un conjunto dinámico dos veces sin modificar el conjunto entre las solicitudes, obtengamos la misma respuesta en ambas ocasiones. Otras aplicaciones pueden requerir una regla preespecificada para elegir el representante, como elegir el miembro más pequeño del conjunto (suponiendo, por supuesto, que los elementos se pueden ordenar).

Como en las otras implementaciones de conjuntos dinámicos que hemos estudiado, representamos cada elemento de un conjunto por un objeto. Dejando que  $x$  denote un objeto, deseamos admitir las siguientes operaciones:

**MAKE-SET.x/** crea un nuevo conjunto cuyo único miembro (y por lo tanto representante) es x. Dado que los conjuntos son disjuntos, requerimos que x no esté ya en algún otro conjunto.

**UNIÓN.x/ y/** une los conjuntos dinámicos que contienen x e y, digamos Sx y Sy, en un nuevo conjunto que es la unión de estos dos conjuntos. Suponemos que los dos conjuntos son disarticulación antes de la operación. El representante del conjunto resultante es cualquier miembro de Sx [ Sy, aunque muchas implementaciones de UNION eligen específicamente al representante de Sx o Sy como el nuevo representante. Dado que requerimos que los conjuntos de la colección sean disjuntos, conceptualmente destruimos los conjuntos Sx y Sy, eliminándolos de la colección S. En la práctica, a menudo absorbemos los elementos de uno de los conjuntos en el otro conjunto.

**FIND-SET.x/** devuelve un puntero al representante del conjunto (único) que contiene ing x.

A lo largo de este capítulo, analizaremos los tiempos de ejecución de estructuras de datos de conjuntos disjuntos en términos de dos parámetros: n, el número de operaciones MAKE-SET , y m, el número total de operaciones MAKE-SET, UNION y FIND-SET .. Dado que los conjuntos son disjuntos, cada operación UNION reduce el número de conjuntos en uno.

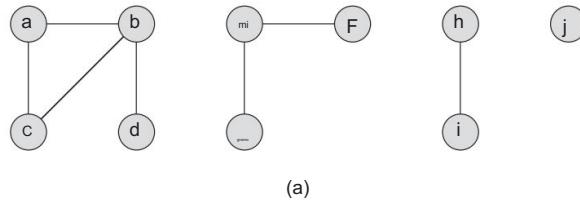
Después de n 1 operaciones UNION , por lo tanto, solo queda un conjunto. El número de operaciones UNION es, por lo tanto, como máximo n 1. Tenga en cuenta también que dado que las operaciones MAKE-SET están incluidas en el número total de operaciones m, tenemos m n. Suponemos que las n operaciones MAKE-SET son las primeras n operaciones realizadas.

#### Una aplicación de estructuras de datos de conjuntos disjuntos

Una de las muchas aplicaciones de las estructuras de datos de conjuntos disjuntos surge al determinar las componentes conexas de un grafo no dirigido (consulte la Sección B.4). La figura 21.1(a), por ejemplo, muestra una gráfica con cuatro componentes conectados.

El procedimiento COMPONENTES-CONECTADOS que sigue utiliza las operaciones de conjuntos disjuntos para calcular los componentes conexos de una gráfica. Una vez que COMPONENTES CONECTADOS ha preprocesado el grafo, el procedimiento MISMO-COMPONENTE responde a consultas sobre si dos vértices están en la misma componente conexa.<sup>1</sup> (En pseudocódigo, denotamos el conjunto de vértices de un grafo por G:V y el conjunto de aristas por G: E.)

<sup>1</sup>Cuando los bordes del gráfico son estáticos, es decir, no cambian con el tiempo, podemos calcular los componentes conectados más rápido usando la búsqueda en profundidad (ejercicio 22.3-12). A veces, sin embargo, los bordes se agregan dinámicamente y necesitamos mantener los componentes conectados a medida que se agrega cada borde. En este caso, la implementación dada aquí puede ser más eficiente que ejecutar una nueva búsqueda en profundidad para cada nuevo borde.



Borde procesado	Colección de conjuntos					
conjuntos iniciales	{a}	{b}	disjuntos {c} {d} {e} {g} {f} {h} {i} {j}			
(b, d)	{a}	{b, d}	{c} {e,g} {e,g} {e,g} {e,f}	{i} {h} {i}		
(e, g)	{a}	{b, d}	g} {e, f, g}	{f}	{h} {i} {h,i}	
(a, c)	{a,c}	{b, d}		{f}	{h,i} {h,i}	
(h, i)	{a,c}	{b, d}		{f}	{h,i}	
(a, b)	{a,b,c,d}			{f}		
(e, f)	{a,b,c,d}					
(b, c)	{a,b,c,d}					

(b)

Figura 21.1 (a) Un gráfico con cuatro componentes conectados: fa; b; C; dg, fe; F; gg, fh; ig y fj g. (b) La colección de conjuntos disjuntos después de procesar cada borde.

CONNECTED-COMPONENTS.G/ 1 para

cada vértice 2 G:V 2

MAKE-SET./ 3 por

cada arista .u; / 2 G:E 4 si FIND-

SET.u/  $\bowtie$  FIND-SET./ UNION.u; /

5

MISMO-COMPONENTE.u; / 1 si

FIND-SET.u/ == FIND-SET./ 2 devuelve

VERDADERO 3 de lo

contrario devuelve FALSO

El procedimiento COMPONENTES-CONECTADOS inicialmente coloca cada vértice en su propio conjunto. Entonces, para cada arista .u; /, une los conjuntos que contienen u y . Según el ejercicio 21.1-2, después de procesar todas las aristas, dos vértices están en la misma componente conexa si y solo si los objetos correspondientes están en el mismo conjunto.

Por lo tanto, COMPONENTES-CONECTADOS calcula conjuntos de tal manera que el procedimiento MISMO-COMPONENTE puede determinar si dos vértices están en la misma dirección.

componente conectado. La figura 21.1(b) ilustra cómo COMPONENTES-CONECTADOS calcula los conjuntos disjuntos.

En una implementación real de este algoritmo de componentes conectados, las representaciones del gráfico y la estructura de datos del conjunto disjunto necesitarían referenciarse entre sí. Es decir, un objeto que representa un vértice contendría un puntero al correspondiente objeto del conjunto disjunto y viceversa. Estos detalles de programación dependen del lenguaje de implementación y no los abordaremos más aquí.

### Ejercicios

#### 21.1-1

Suponga que CONNECTED-COMPONENTS se ejecuta en el grafo no dirigido  $G = \langle V, E \rangle$ , donde  $V = \{a, b, C, d, mi, F, gramo, h, i, j, kg\}$  y los bordes de  $E$  se procesan en el orden  $\{d, i, /, F, k, .gramo, i, /, .b, gramo, .a, h, .i, j, .d, k, .b, j, .d, f, .gramo, j, .a, mi\}$ . Haz una lista de los vértices en cada componente conectado después de cada iteración de las líneas 3 a 5.

#### 21.1-2

Muestre que después de que COMPONENTES-CONECTADOS procese todas las aristas, dos vértices están en el mismo componente conexo si y sólo si están en el mismo conjunto.

#### 21.1-3

Durante la ejecución de CONNECTED-COMPONENTS en un gráfico no dirigido  $G = \langle V, E \rangle$  con  $k$  componentes conexas, ¿cuántas veces se llama FIND-SET? ¿Cuántas veces se llama UNION? Exprese sus respuestas en términos de  $|V|$ ,  $|E|$  y  $k$ .

## 21.2 Representación de listas enlazadas de conjuntos disjuntos

La figura 21.2(a) muestra una forma sencilla de implementar una estructura de datos de conjuntos disjuntos: cada conjunto está representado por su propia lista enlazada. El objeto de cada conjunto tiene los atributos cabeza, que apunta al primer objeto de la lista, y cola, que apunta al último objeto. Cada objeto de la lista contiene un miembro del conjunto, un puntero al siguiente objeto de la lista y un puntero al objeto del conjunto. Dentro de cada lista enlazada, los objetos pueden aparecer en cualquier orden. El representante es el miembro del conjunto en el primer objeto de la lista.

Con esta representación de lista enlazada, tanto CREAR CONJUNTO como ENCONTRAR CONJUNTO son fáciles y requieren  $O(1)$  tiempo. Para realizar MAKE-SET. $x$ , creamos una nueva lista enlazada cuyo único objeto es  $x$ . Para FIND-SET. $x$ , simplemente seguimos el puntero desde  $x$  hasta su objeto establecido y luego devolvemos el miembro en el objeto al que apunta la cabeza. Por ejemplo, en la figura 21.2(a), la llamada FIND-SET. $g$  devolvería  $f$ .

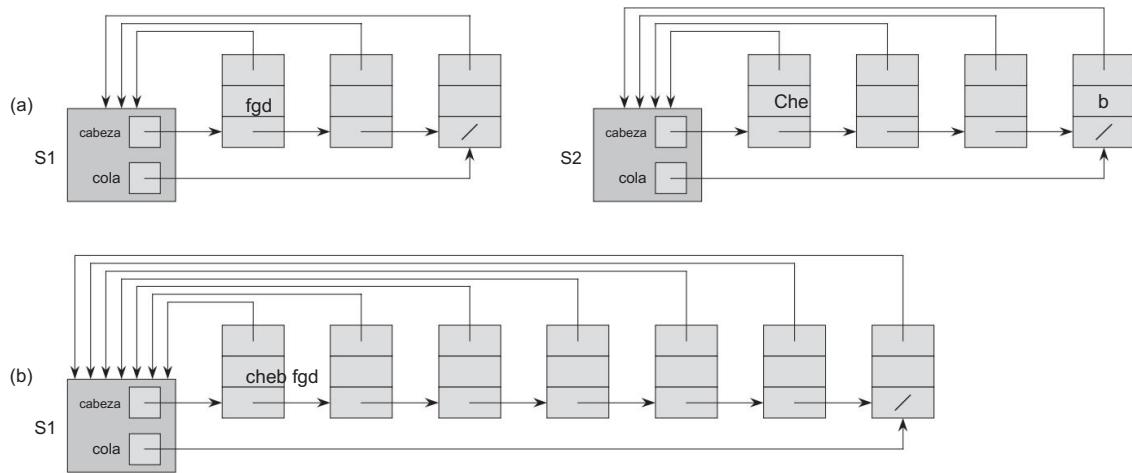


Figura 21.2 (a) Representaciones de listas enlazadas de dos conjuntos. El conjunto S1 contiene los miembros d, f y g, con el representante f, y el conjunto S2 contiene los miembros b, c, e y h, con el representante c. Cada objeto de la lista contiene un miembro del conjunto, un puntero al siguiente objeto de la lista y un puntero al objeto del conjunto. Cada objeto establecido tiene punteros cabeza y cola al primer y último objeto, respectivamente. (b) El resultado de UNION.g; e/, que añade la lista enlazada que contiene e a la lista enlazada que contiene g. El representante del conjunto resultante es f . El objeto establecido para la lista de e, S2, se destruye.

Una implementación simple de la unión.

La implementación más simple de la operación UNION utilizando la representación de conjunto de listas enlazadas toma mucho más tiempo que MAKE-SET o FIND-SET. Como muestra la figura 21.2(b), realizamos UNION.x; y/ agregando la lista de y al final de la lista de x. El representante de la lista de x se convierte en el representante del conjunto resultante. Usamos el puntero de cola para la lista de x para encontrar rápidamente dónde agregar la lista de y. Debido a que todos los miembros de la lista de y se unen a la lista de x, podemos destruir el objeto establecido para la lista de y. Desafortunadamente, debemos actualizar el puntero al objeto establecido para cada objeto originalmente en la lista de y, lo que toma el tiempo lineal en la longitud de la lista de y. En la Figura 21.2, por ejemplo, la operación UNION.g; e/ hace que los punteros se actualicen en los objetos para b, c, e y h.

De hecho, podemos construir fácilmente una secuencia de m operaciones en n objetos que requiere  $n^2$  tiempo. Supongamos que tenemos objetos  $x_1; x_2; \dots; x_n$ . Ejecutamos la secuencia de n operaciones MAKE-SET seguidas de n-1 operaciones UNION que se muestran en la figura 21.3, de modo que  $m = 2n - 1$ . Pasamos  $n^2$  tiempo realizando las n operaciones MAKE-SET . Debido a que la i-ésima operación UNION actualiza i objetos, el número total de objetos actualizados por todas las n-1 operaciones UNION es

Operación	Número de objetos actualizados
MAKE-SET.x1/	1
MAKE-SET.x2/	1
⋮	⋮
MAKE-SET.xn/	1
UNIÓN.x2; x1/	1 1
UNIÓN.x3; x2/	2
UNIÓN.x4; x3/	3
⋮	⋮
UNIÓN.xn; xn1/	n 1

Figura 21.3 Una secuencia de  $2n - 1$  operaciones en  $n$  objetos que toma  $.n^2/$  tiempo, o  $.n/n$  tiempo por operación en promedio, utilizando la representación de conjunto de lista enlazada y la implementación simple de UNION.

Xn1 yo D ,.n2/ :  
ID1

El número total de operaciones es  $2n - 1$ , por lo que cada operación en promedio requiere  $.n/n$  tiempo. Es decir, el tiempo amortizado de una operación es  $.n/n$ .

#### Una heurística de unión ponderada

En el peor de los casos, la implementación anterior del procedimiento UNION requiere un promedio de  $.n/n$  tiempo por llamada porque podemos agregar una lista más larga a una lista más corta; debemos actualizar el puntero al objeto establecido para cada miembro de la lista más larga. Supongamos, en cambio, que cada lista también incluye la longitud de la lista (que podemos mantener fácilmente) y que siempre agregamos la lista más corta a la más larga, rompiendo los vínculos arbitrariamente. Con esta simple heurística de unión ponderada, una sola operación UNION aún puede tomar  $.n/n$  tiempo si ambos conjuntos tienen  $n/n$  miembros. Sin embargo, como muestra el siguiente teorema, una secuencia de  $m$  operaciones MAKE-SET, UNION y FIND-SET,  $n$  de las cuales son operaciones MAKE-SET, toma  $O(n \lg n)$  tiempo.

#### Teorema 21.1

Usando la representación de lista enlazada de conjuntos disjuntos y la heurística de unión ponderada, una secuencia de  $m$  operaciones CREAR-CONJUNTO, UNION y ENCONTRAR-CONJUNTO,  $n$  de las cuales son operaciones CREAR-CONJUNTO, toma  $O(n \lg n)$  tiempo.

Prueba Debido a que cada operación UNION une dos conjuntos disjuntos, realizamos como máximo  $n_1$  operaciones UNION en total. Ahora limitamos el tiempo total que tardan estas operaciones UNION . Comenzamos determinando, para cada objeto, un límite superior en la cantidad de veces que se actualiza el puntero del objeto de regreso a su objeto establecido. Considere un objeto particular  $x$ . Sabemos que cada vez que se actualizó el puntero de  $x$ ,  $x$  debe haber comenzado en el conjunto más pequeño. Por lo tanto, la primera vez que se actualizó el puntero de  $x$ , el conjunto resultante debe haber tenido al menos 2 miembros. De manera similar, la próxima vez que se actualice el puntero de  $x$ , el conjunto resultante debe haber tenido al menos 4 miembros. Continuando, observamos que para cualquier  $k_n$ , después de que el puntero de  $x$  ha sido actualizado  $dlg$  veces, el conjunto resultante debe tener al menos  $k$  miembros. Dado que el conjunto más grande tiene como máximo  $n$  miembros, el puntero de cada objeto se actualiza como máximo  $dlg$  veces en todas las operaciones UNION . Por lo tanto, el tiempo total dedicado a actualizar los punteros de objetos en todas las operaciones UNION es  $O(n \lg n)$ . También debemos tener en cuenta la actualización de los punteros de cola y las longitudes de la lista, que tardan solo  $O(1)$  vez por operación UNION . El tiempo total empleado en todas las operaciones UNION es, por tanto,  $O(n \lg n)$ .

El tiempo para la secuencia completa de  $m$  operaciones sigue fácilmente. Cada operación MAKE SET y FIND-SET toma  $O(1)$  tiempo, y hay  $O(m)$  de ellos. El tiempo total para toda la secuencia es, por tanto,  $O(m \lg n)$ . ■

## Ejercicios

### 21.2-1

Escriba un pseudocódigo para MAKE-SET, FIND-SET y UNION usando la representación de lista enlazada y la heurística de unión ponderada. Asegúrese de especificar los atributos que asume para los objetos de conjunto y los objetos de lista.

### 21.2-2

Muestre la estructura de datos que resulta y las respuestas devueltas por las operaciones FIND-SET en el siguiente programa. Utilice la representación de lista enlazada con la heurística de unión ponderada.

```

1 para i D 1 a 16 2
MAKE-SET.xi/ 3 para i D 1 a
15 por 2 UNION.xi; xiC1/ 4 5
para i D 1 a 13 por
4 UNION.xi; xiC2/ 6 7
UNIÓN.x1; x5/ 8
UNIÓN.x11; x13/ 9
UNIÓN.x1; x10/ 10
ENCONTRAR-
CONJUNTO.x2/ 11
ENCONTRAR-CONJUNTO.x9/

```

Suponga que si los conjuntos que contienen  $x_i$  y  $x_j$  tienen el mismo tamaño, entonces la operación UNION. $x_i$ ;  $x_j$  / añade la lista de  $x_j$  a la lista de  $x_i$ .

#### 21.2-3

Adapte la prueba agregada del teorema 21.1 para obtener límites de tiempo amortizados de  $O(1/n)$  para MAKE-SET y FIND-SET y  $O(\lg n)$  para UNION utilizando la representación de lista enlazada y la heurística de unión ponderada.

#### 21.2-4

Proporcione un límite asintótico ajustado en el tiempo de ejecución de la secuencia de operaciones de la figura 21.3, suponiendo la representación de lista enlazada y la heurística de unión ponderada.

#### 21.2-5

El profesor Gompers sospecha que podría ser posible mantener solo un puntero en cada objeto del conjunto, en lugar de dos (cabeza y cola), mientras se mantiene en dos el número de punteros en cada elemento de la lista. Muestre que la sospecha del profesor está bien fundada describiendo cómo representar cada conjunto mediante una lista enlazada de modo que cada operación tenga el mismo tiempo de ejecución que las operaciones descritas en esta sección. Describa también cómo funcionan las operaciones. Su esquema debe permitir la heurística de unión ponderada, con el mismo efecto que se describe en esta sección. (Sugerencia: use la cola de una lista enlazada como representante de su conjunto).

#### 21.2-6

Sugiera un cambio simple al procedimiento UNION para la representación de lista enlazada que elimine la necesidad de mantener el puntero de cola en el último objeto de cada lista. Ya sea que se use o no la heurística de unión ponderada, su cambio no debería cambiar el tiempo de ejecución asintótico del procedimiento UNION . (Sugerencia: en lugar de agregar una lista a otra, únalas).

---

### 21.3 Bosques disjuntos

En una implementación más rápida de conjuntos disjuntos, representamos conjuntos por árboles con raíz, con cada nodo que contiene un miembro y cada árbol que representa un conjunto. En un bosque de conjuntos disjuntos, ilustrado en la figura 21.4(a), cada miembro apunta solo a su parente. La raíz de cada árbol contiene el representante y es su propio parente. Como veremos, aunque los algoritmos sencillos que usan esta representación no son más rápidos que los que usan la representación de lista enlazada, al introducir dos heurísticas —“unión por rango” y “compresión de ruta”— podemos lograr una disjunción asintóticamente óptima. -establecer la estructura de datos.

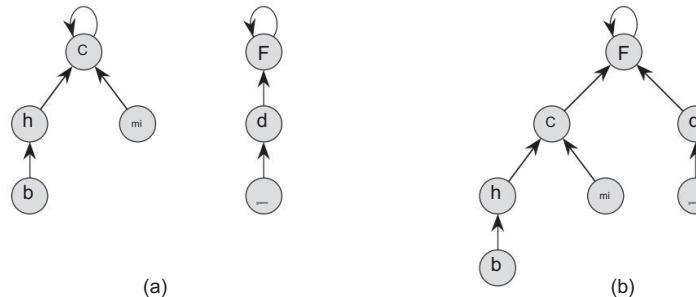


Figura 21.4 Un bosque inconexo. (a) Dos árboles que representan los dos conjuntos de la figura 21.2. El árbol de la izquierda representa el conjunto  $fb; C; mi$ ;  $hg$ , con  $c$  como representante, y el árbol de la derecha representa el conjunto  $fd; F; gg$ , con  $f$  como representante. (b) El resultado de  $\text{UNION.e}; \text{gramo}/$ .

Realizamos las tres operaciones de conjuntos disjuntos de la siguiente manera. Una operación MAKE-SET simplemente crea un árbol con un solo nodo. Realizamos una operación FIND-SET siguiendo los punteros principales hasta encontrar la raíz del árbol. Los nodos visitados en este camino simple hacia la raíz constituyen el camino de búsqueda. La operación AUNION , que se muestra en la figura 21.4(b), hace que la raíz de un árbol apunte a la raíz del otro.

#### Heurísticas para mejorar el tiempo de ejecución

Hasta ahora, no hemos mejorado la implementación de la lista enlazada. Una secuencia de  $n^1$  operaciones UNION puede crear un árbol que es solo una cadena lineal de  $n$  nodos. Sin embargo, al usar dos heurísticas, podemos lograr un tiempo de ejecución que es casi lineal en el número total de operaciones  $m$ .

La primera heurística, unión por rango, es similar a la heurística de unión ponderada que usamos con la representación de lista enlazada. El enfoque obvio sería hacer que la raíz del árbol con menos nodos apunte a la raíz del árbol con más nodos.

En lugar de realizar un seguimiento explícito del tamaño del subárbol enraizado en cada nodo, utilizaremos un enfoque que facilita el análisis. Para cada nodo, mantenemos un rango, que es un límite superior en la altura del nodo. En unión por rango, hacemos que la raíz con menor rango apunte a la raíz con mayor rango durante una operación UNION .

La segunda heurística, la compresión de caminos, también es bastante simple y altamente efectiva. Como se muestra en la Figura 21.5, lo usamos durante las operaciones FIND-SET para hacer que cada nodo en el camino de búsqueda apunte directamente a la raíz. La compresión de ruta no cambia ningún rango.

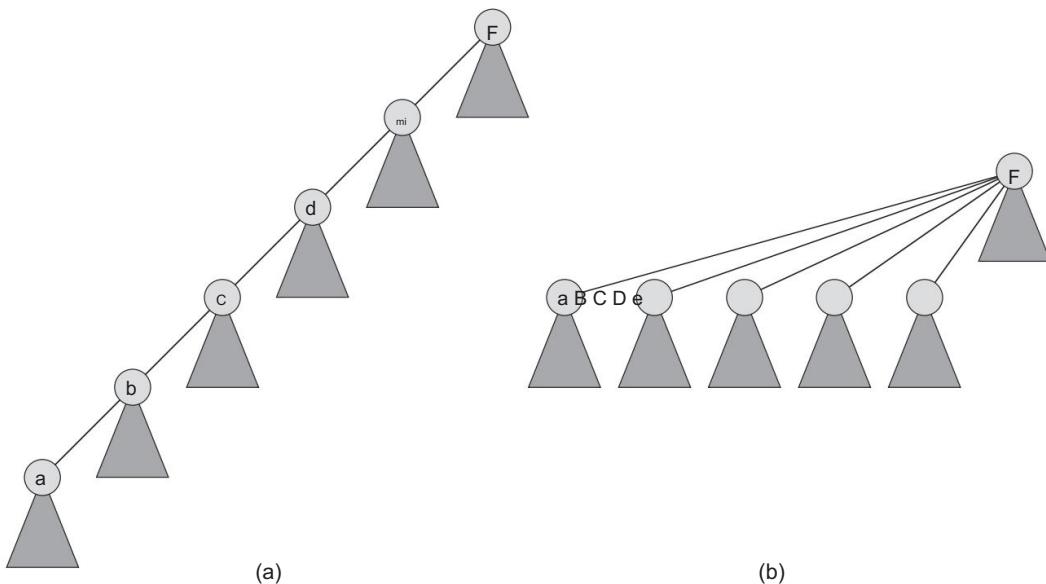


Figura 21.5 Compresión de trayectoria durante la operación FIND-SET. Se omiten las flechas y los bucles automáticos en las raíces. (a) Un árbol que representa un conjunto antes de ejecutar FIND-SET.a/. Los triángulos representan subárboles cuyas raíces son los nodos que se muestran. Cada nodo tiene un puntero a su padre. (b) El mismo conjunto después de ejecutar FIND-SET.a/. Cada nodo en la ruta de búsqueda ahora apunta directamente a la raíz.

#### Pseudocódigo para bosques de conjuntos disjuntos

Para implementar un bosque de conjuntos disjuntos con la heurística de unión por rango, debemos realizar un seguimiento de los rangos. Con cada nodo  $x$ , mantenemos el valor entero  $x: \text{rango}$ , que es un límite superior en la altura de  $x$  (el número de aristas en el camino simple más largo entre  $x$  y una hoja descendiente). Cuando MAKE-SET crea un conjunto singleton, el único nodo en el árbol correspondiente tiene un rango inicial de 0. Cada operación FIND-SET deja todos los rangos sin cambios. La operación UNION tiene dos casos, dependiendo de si las raíces de los árboles tienen el mismo rango. Si las raíces tienen un rango diferente, hacemos que la raíz con un rango más alto sea el parente de la raíz con un rango más bajo, pero los rangos mismos permanecen sin cambios. Si, en cambio, las raíces tienen rangos iguales, elegimos arbitrariamente una de las raíces como parente e incrementamos su rango.

Pongamos este método en pseudocódigo. Designamos al parente del nodo  $x$  por  $x:p$ . El procedimiento LINK, una subrutina llamada por UNION, toma punteros a dos raíces como entradas.

MAKE-SET.x/ 1

x:p D x 2  
x:rango D 0

UNIÓN.x; y/ 1

ENLACE.FIND-SET.x/; FIND-SET.y//

ENLACE.x;

y/ 1 si x:rango > y:rango 2  
y:p D x 3 else x:p D  
y 4 si x:rango ==  
y:rango y:rango D y:rango C 1  
5

El procedimiento FIND-SET con compresión de ruta es bastante simple:

FIND-SET.x/ 1

si x ≠ x:p 2 x:p  
D FIND-SET.x:p/ 3 devuelve x:p

El procedimiento FIND-SET es un método de dos pasadas: cuando recurre, hace una pasada por la ruta de búsqueda para encontrar la raíz, y cuando la recursión se desenrolla, hace una segunda pasada por la ruta de búsqueda para actualizar cada nodo a apuntar directamente a la raíz. Cada llamada de FIND-SET.x/ devuelve x:p en la línea 3. Si x es la raíz, entonces FIND-SET omite la línea 2 y en su lugar devuelve x:p, que es x; este es el caso en el que la recursividad toca fondo. De lo contrario, se ejecuta la línea 2 y la llamada recursiva con el parámetro x:p devuelve un puntero a la raíz. La línea 2 actualiza el nodo x para que apunte directamente a la raíz y la línea 3 devuelve este puntero.

Efecto de las heurísticas en el tiempo de ejecución

Por separado, la unión por rango o la compresión de rutas mejora el tiempo de ejecución de las operaciones en bosques de conjuntos disjuntos, y la mejora es aún mayor cuando usamos las dos heurísticas juntas. Solo, la unión por rango produce un tiempo de ejecución de  $O(m \lg n)$  (vea el ejercicio 21.4-4), y este límite es estrecho (vea el ejercicio 21.3-3).

Aunque no lo demostraremos aquí, para una secuencia de  $n$  operaciones MAKE-SET (y, por lo tanto, como máximo  $n$  operaciones UNION) y  $f$  operaciones FIND-SET, la heurística de compresión de ruta por sí sola da un tiempo de ejecución en el peor de los casos de  $C f .1 C \log_2 C f = n //$ .

Cuando usamos tanto la unión por rango como la compresión de ruta, el tiempo de ejecución del peor de los casos es  $O(m \cdot n^k)$ , donde  $\cdot n^k$  es una función de crecimiento muy lento, que definimos en la Sección 21.4. En cualquier aplicación concebible de una estructura de datos de conjuntos disjuntos,  $\cdot n^k / 4$ ; por lo tanto, podemos ver el tiempo de ejecución como lineal en  $m$  en todas las situaciones prácticas. Estrictamente hablando, sin embargo, es superlineal. En la Sección 21.4 demostramos esta cota superior.

### Ejercicios

#### 21.3-1

Vuelva a hacer el ejercicio 21.2-2 usando un bosque de conjuntos disjuntos con unión por rango y compresión de caminos.

#### 21.3-2

Escriba una versión no recursiva de FIND-SET con compresión de ruta.

#### 21.3-3

Proporcione una secuencia de  $m$  operaciones MAKE-SET, UNION y FIND-SET ,  $n$  de las cuales son operaciones MAKE-SET , que toman  $m \lg n$  tiempo cuando usamos unión por rango solamente.

#### 21.3-4

Suponga que deseamos agregar la operación PRINT-SET. $x$ /, que recibe un nodo  $x$  e imprime todos los miembros del conjunto de  $x$ , en cualquier orden. Muestre cómo podemos agregar un solo atributo a cada nodo en un bosque de conjunto disjunto para que PRINT-SET. $x$ / tome el tiempo lineal en el número de miembros del conjunto de  $x$  y los tiempos de ejecución asintóticos de las otras operaciones no cambien. Suponga que podemos imprimir cada miembro del conjunto en  $O(1)$  tiempo.

#### 21.3-5 ?

Muestre que cualquier secuencia de  $m$  operaciones MAKE-SET, FIND-SET y LINK , donde todas las operaciones LINK aparecen antes que cualquiera de las operaciones FIND-SET , toma solo  $O(m/\log m)$  tiempo si usamos compresión de caminos y unión por rango. ¿Qué sucede en la misma situación si usamos solo la heurística de compresión de rutas?

#### 21.4 Análisis de unión por rango con compresión de trayectoria

Como se señaló en la Sección 21.3, la heurística combinada de unión por rango y compresión de trayectoria se ejecuta en el tiempo  $O(m \cdot n^{\frac{1}{2}})$  para  $m$  operaciones de conjunto disjunto en  $n$  elementos. En esta sección, examinaremos la función  $\alpha$  para ver cuán lentamente crece. Luego demostraremos este tiempo de ejecución usando el método potencial de análisis amortizado.

Una función de crecimiento muy rápido y su inversa de crecimiento muy lento

Para enteros  $k \geq 0$  y  $j \in \mathbb{N}$ , definimos la función  $A_k(j)$  as

si k D 0

$$A_{k,j}/D(j) \in A_{j,C1} / j \text{ si } k=1$$

donde la expresión  $A.iC1_{f,i}.$  usa la notación de iteración funcional dada en la Sección

3.2. Específicamente,  $A_0 / k_{1,i} / D_i$  y  $A_i / k_{1,i} / D_{k_1 A_i j_1} / j_1$  para  $i \in I$ .

Nos referiremos al parámetro  $k$  como el nivel de la función A.

La función  $Ak.j$  / crece estrictamente tanto con  $j$  como con  $k$ . Para ver qué tan rápido crece esta función, primero obtenemos expresiones de forma cerrada para  $A1.i$  / y  $A2.i$  /.

Lema 21.2

Para cualquier entero  $i \geq 1$ , tenemos  $A_{1,i} / D_{2,i} \leq C_1$ .

Prueba Primero usamos inducción en  $i$  para mostrar que  $A_i \vdash j \rightarrow D_j \wedge C_i$ . Para el caso base, tenemos  $\alpha \vdash i \rightarrow D_i \wedge C_0$ . Para el paso inductivo, suponga que  $A_i \vdash i \rightarrow D_i \wedge C_i$ .

C.i.j // D Econtrase A1j/Dj//D A1jAlmamente, notamos que A1.j / D A.jC1 . j / D j C . j C 1 / D 2j C 1.

Lema 21.3

Para cualquier entero  $j \geq 1$ , tenemos  $A_2(j) / D \geq C_1(j) C_1(j)$

Prueba Primero usamos inducción en  $i$  para mostrar que,  $A_i / .j / D \ 2i . j \ C \ 1 / el$  1. para caso base, tenemos  $A.0/ .j / D \ 20.j \ C \ 1 / 1.$  Para el paso inductivo , suponga que  $A.i/ 1.$  Entonces  $A_i / .j / D \ 2i . j \ C \ 1 / el$

Finalmente, notamos que  $A_2.i / D A_i C_1 / .i / D 2iC_1.i C 1 / 1.$

Ahora podemos ver qué tan rápido crece  $Ak.j$ / simplemente examinando  $Ak.1/$  para los niveles  $k \in D$ : 0; 1; 2; 3; 4. A partir de la definición de  $A0.k/$  y los lemas anteriores, tenemos  $A0.1/D \subseteq C_1 \cap D_2$ ;  $A1.1/D \subseteq C_2 \cap D_3$  y  $A2.1/D \subseteq C_1 \cap C_2 \cap D_7$ .

También tenemos

A3.1/ D A.2/ .1/ D  
 A2.A2.1// D  
 A2.7/ D  
 28 8 1 D 211  
 1  
 D 2047

y

A4.1/ D A.2/ .1/  
 D A3.A3.1// D  
 A3.2047/ D  
 A.2048/ .2047/  
 A2.2047/  
 D 22048 2048 1 >  
 22048 D  
 .24/ 512 D  
 16512  
 1080

que es el número estimado de átomos en el universo observable. (El símbolo " $\gg$ " denota la relación "mucho mayor que".)

Definimos la inversa de la función  $A_k.n/$ , para entero  $n \geq 0$ , por

$\lfloor n/ \rfloor \leq k \leq \lceil n/ \rceil$

En palabras,  $\lfloor n/ \rfloor$  es el nivel más bajo  $k$  para el cual  $A_k.1/ \geq n$ . De los valores anteriores de  $A_k.1/$ , vemos que

$\lfloor n/ \rfloor =$ 1 para $n \leq 3$ $\lfloor n/ \rfloor =$ 2 para $4 \leq n \leq 7$ $\lfloor n/ \rfloor =$ 3 para $8 \leq n \leq 2047$ $\lfloor n/ \rfloor =$ 4 para $2048 \leq n \leq 1080$	$\vdots$ $\vdots$ $\vdots$ $\vdots$
--	--

Es solo para valores de  $n$  tan grandes que el término "astronómico" los subestima (mayores que A4.1/, un número enorme) que  $\lfloor n/ \rfloor > 4$ , y por lo tanto  $\lfloor n/ \rfloor / 4$  para todos los propósitos prácticos.

### Propiedades de los rangos

En el resto de esta sección, demostramos una cota  $\text{Om}_{\cdot n//}$  en el tiempo de ejecución de las operaciones de conjuntos disjuntos con unión por rango y compresión de trayectoria. Para probar este límite, primero demostramos algunas propiedades simples de los rangos.

#### Lema 21.4

Para todos los nodos  $x$ , tenemos  $x:\text{rango } x:p:\text{rango}$ , con desigualdad estricta si  $x \neq x:p$ .

El valor de  $x:\text{rank}$  es inicialmente 0 y aumenta con el tiempo hasta que  $x \neq x:p$ ; a partir de ese momento,  $x:\text{rank}$  no cambia. El valor de  $x:p:\text{rank}$  aumenta monótonamente con el tiempo.

Prueba La prueba es una inducción directa sobre el número de operaciones, usando las implementaciones de MAKE-SET, UNION y FIND-SET que aparecen en la Sección 21.3. Lo dejamos como Ejercicio 21.4-1. ■

#### Corolario 21.5 A

medida que seguimos el camino simple desde cualquier nodo hacia una raíz, los rangos de los nodos aumentan estrictamente. ■

#### Lema 21.6 Todo

nodo tiene rango a lo sumo  $n$ .

Prueba El rango de cada nodo comienza en 0 y aumenta solo con las operaciones de ENLACE .

Como hay como máximo  $n$  1 operaciones UNION , también hay como máximo  $n$  1 operaciones LINK . Debido a que cada operación LINK deja todos los rangos solos o aumenta el rango de algún nodo en 1, todos los rangos son como máximo  $n$  1. ■

El lema 21.6 proporciona un límite débil en los rangos. De hecho, cada nodo tiene rango a lo sumo  $\text{blg } nc$  (vea el Ejercicio 21.4-2). Sin embargo, el límite más flexible del Lema 21.6 será suficiente para nuestros propósitos.

#### Demostrando el límite de tiempo

Usaremos el método potencial de análisis amortizado (vea la Sección 17.3) para probar el límite de tiempo de  $\text{Om}_{\cdot n//}$ . Al realizar el análisis amortizado, resultará conveniente suponer que invocamos la operación ENLACE en lugar de la operación UNIÓN . Es decir, dado que los parámetros del procedimiento ENLACE apuntan a dos raíces, actuamos como si realizáramos las operaciones FIND-SET apropiadas por separado. El siguiente lema muestra que incluso si contamos las operaciones FIND-SET adicionales inducidas por las llamadas UNION , el tiempo de ejecución asintótico permanece sin cambios.

**Lema 21.7**

Supongamos que convertimos una secuencia  $S_0$  de  $m_0$  operaciones MAKE-SET, UNION y FIND-SET en una secuencia  $S$  de  $m$  operaciones MAKE-SET, LINK y FIND-SET convirtiendo cada UNION en dos operaciones FIND-SET seguidas por un ENLACE. Entonces, si la secuencia  $S$  se ejecuta en el tiempo  $O(m \cdot n)$ , la secuencia  $S_0$  se ejecuta en el tiempo  $O(m_0 \cdot n)$ .

Prueba Dado que cada operación UNION en la secuencia  $S_0$  se convierte en tres operaciones en  $S$ , tenemos  $m_0 \leq 3m$ . Dado que  $m \leq O(m_0)$ , un límite de tiempo  $O(m \cdot n)$  para la secuencia convertida  $S$  implica un límite de tiempo  $O(m_0 \cdot n)$  para la secuencia original  $S_0$ .

■

En el resto de esta sección, supondremos que la secuencia inicial de  $m_0$  operaciones MAKE-SET, UNION y FIND-SET se ha convertido en una secuencia de  $m$  operaciones MAKE-SET, LINK y FIND-SET. Ahora demostramos un límite de tiempo  $O(m \cdot n)$  para la secuencia convertida y recurrimos al Lema 21.7 para demostrar el tiempo de ejecución  $O(m_0 \cdot n)$  de la secuencia original de  $m_0$  operaciones.

**Función potencial**

La función de potencial que usamos asigna un potencial  $q_x$  a cada nodo  $x$  en el bosque de conjuntos disjuntos después de  $q$  operaciones. Sumamos los potenciales de nodo para el potencial de todo el bosque:  $\hat{q}_D q_x$ , donde  $\hat{q}$  denota el potencial del bosque después de  $q$  operaciones. El bosque está vacío antes de la primera operación y establecemos arbitrariamente  $\hat{0} = 0$ . Ningún potencial  $\hat{q}$  será negativo.

El valor de  $q_x$  depende de si  $x$  es una raíz de árbol después de la operación  $q$ -ésima.

Si lo es, o si  $x:p:rango D 0$ , entonces  $q_x = D \cdot n / x:rango$ .

Ahora suponga que después de la operación  $q$ -ésima,  $x$  no es una raíz y que  $x:rango = 1$ . Necesitamos definir dos funciones auxiliares en  $x$  antes de que podamos definir  $q_x$ . Primero definimos

$level.x = \max_{k \in W} \text{rank}(A_k \cdot x:rango) : g$

Es decir,  $level.x$  es el mayor nivel  $k$  para el cual  $A_k$  aplicado al rango de  $x$ , no es mayor que el rango del padre de  $x$ .

Afirmamos que

$$0 \leq level.x < n ; \quad (21.1)$$

que vemos de la siguiente manera. Tenemos

$$x:p:rango = x:rango C 1 \quad (\text{por el Lema 21.4})$$

$$D A_0 \cdot x:rango = (A_0 \cdot j) ,$$

lo que implica que  $level.x = 0$ , y tenemos

$A_{\cdot,n}/x:\text{rango}/$   $A_{\cdot,n}/1/$  (porque  $Ak.j$  / es estrictamente creciente) (según  
la definición de  $\cdot,n/ > x:p:\text{rank}$   
(por el Lema 21.6) ,

lo que implica que  $\text{level.x} < \text{.n.}$ . Tenga en cuenta que debido a que  $x:p:rank$  aumenta monótonamente con el tiempo, también lo hace  $\text{level.x}.$ .

La segunda función auxiliar se aplica cuando x:rango 1:

iter.x/ D max ° i W x:p:rank Ai / level.x/.x:rank/ :

Es decir,  $\text{iter.x/}$  es el mayor número de veces que podemos aplicar iterativamente  $\text{Alevel.x/}$ , aplicado inicialmente al rango de  $x$ , antes de obtener un valor mayor que el rango del padre de  $x$ .

Decimos que cuando  $x$ : rango 1, tenemos

1 iter.x/ x:range : (21.2)

que vemos de la siguiente manera. Tenemos

x:p:rango Aleyel.x/.x:rank/ (por definición de leylel.x/)

D A.1/ `|level.x/x:rank|` (por definición de iteración funcional) .

lo que implica que  $\text{iter.x}/$

Ax:rankC1/.x:rank/ D Alevel.x/C1.x:rank/ (por definición de Ak.j /) level.x/ > x:p:rank  
 (por definición de level.x/).

lo que implica que  $\text{iter.x} / \text{x.rank}$ . Tenga en cuenta que debido a que  $\text{x.p.rank}$  aumenta monótonamente con el tiempo, para que  $\text{iter.x} / \text{disminuya}$ ,  $\text{level.x} / \text{debe aumentar}$ . Mientras  $\text{level.x} / \text{permanezca sin cambios}$ ,  $\text{iter.x} / \text{debe aumentar o permanecer sin cambios}$ .

Con estas funciones auxiliares en su lugar, estamos listos para definir el potencial de nodo x después de q operaciones:

si  $x$  es una raíz o  $x$  rango D C

gx/ D ( .n/ x:rank n/ level.x//x:rank iter.x/ si x no es raíz y x:rank 1

A continuación investigamos algunas propiedades útiles de los potenciales de nodo.

### Lema 21.8

Para cada nodo  $x$ , y para todos los recuentos de operaciones  $q$ , tenemos

0  $\alpha x/\beta n/\chi \text{range}$

Prueba Si  $x$  es una raíz o  $x:\text{rango } D \neq 0$ , entonces  $qx/\text{D}_{\cdot,n}/x:\text{rango}$  por definición.  
 Supongamos ahora que  $x$  no es una raíz y que  $x:\text{rango } 1$ . Obtenemos un límite inferior en  $qx/$  maximizando  $\text{level.x/ e iter.x/}$ . Por el límite (21.1),  $\text{level.x/}_{\cdot,n}/1$ , y por el límite (21.2),  $\text{iter.x/ x:rank}$ . De este modo,

$$\begin{aligned} qx/\text{D}_{\cdot,n}/\text{nivel.x/ x:range iter.x/}_{\cdot,n}/\cdot,n/1// \\ & \quad x:\text{range x:range} \\ & \quad D x:\text{range x:range} \\ & \quad D 0 \end{aligned}$$

De manera similar, obtenemos un límite superior en  $qx/$  minimizando  $\text{level.x/ e iter.x/}$ .  
 Por la cota (21.1),  $\text{level.x/}_0$ , y por la cota (21.2),  $\text{iter.x/}$  1. Así,

$$\begin{aligned} qx/ & \quad \cdot,n/0/x:\text{range } 1 D_{\cdot,n}/ \\ & \quad x:\text{range } 1 < \cdot,n/x:\text{range} : \end{aligned}$$

### Corolario 21.9

Si el nodo  $x$  no es una raíz y  $x:\text{rango} > 0$ , entonces  $qx/ < \cdot,n/x:\text{range}$ .

### Cambios potenciales y costos amortizados de operaciones

Ahora estamos listos para examinar cómo las operaciones de conjuntos disjuntos afectan los potenciales de los nodos. Con una comprensión del cambio en el potencial debido a cada operación, podemos determinar el costo amortizado de cada operación.

#### Lema 21.10

Sea  $x$  un nodo que no es raíz y suponga que la operación  $q$ -ésima es un ENLACE o ENCONTRAR-CONJUNTO. Luego, después de la operación  $q$ -ésima,  $qx/q1.x/$ . Además, si  $x:\text{rango } 1$  y  $\text{level.x/ o iter.x/}$  cambia debido a la operación  $q$ -ésima, entonces  $qx/q1.x/$

1. Es decir, el potencial de  $x$  no puede aumentar, y si tiene rango positivo y  $\text{level.x/ o iter.x/}$  cambia, entonces el potencial de  $x$  cae al menos en 1.

Prueba Debido a que  $x$  no es una raíz, la operación  $q$ -ésima no cambia  $x:\text{rank}$ , y debido a que  $n$  no cambia después de las  $n$  operaciones iniciales de MAKE-SET,  $\cdot,n/$  también permanece sin cambios. Por lo tanto, estos componentes de la fórmula para el potencial de  $x$  siguen siendo los mismos después de la operación  $q$ -ésima. Si  $x:\text{rango } D \neq 0$ , entonces  $qx/\text{D}_{q1.x/}D \neq 0$ . Ahora suponga que  $x: \text{rango } 1$ .

Recuerde que  $\text{level.x/}$  aumenta monótonamente con el tiempo. Si la operación  $q$ -ésima deja  $\text{level.x/}$  sin cambios, entonces  $\text{iter.x/}$  aumenta o permanece sin cambios.

Si tanto  $\text{level.x/}$  como  $\text{iter.x/}$  no cambian, entonces  $qx/\text{D}_{q1.x/}$ . Si  $\text{nivel.x/}$

no cambia y  $\text{iter.x/}$  aumenta, luego aumenta en al menos 1, y así  $\text{qx/ q1.x/ 1}$ .

Finalmente, si la operación  $q$ -ésima aumenta  $\text{level.x/}$ , aumenta al menos en 1, de modo que el valor del término  $_{\cdot,n/} \text{level.x// x:rank}$  disminuye al menos en  $x:rank$ . Debido a que  $\text{level.x/}$  aumentó, el valor de  $\text{iter.x/}$  podría caer, pero de acuerdo con el límite (21.2), la caída es como máximo  $x: rango 1$ . Por lo tanto, el aumento en el potencial debido al cambio en  $\text{iter.x/}$  es menor que la disminución de potencial debido al cambio en  $\text{level.x/}$ , y concluimos que  $\text{qx/ q1.x/ 1}$ . ■

Nuestros últimos tres lemas muestran que el costo amortizado de cada operación MAKE-SET, LINK y FIND-SET es  $O_{\cdot,n//}$ . Recuerde de la ecuación (17.2) que el costo amortizado de cada operación es su costo real más el aumento de potencial debido a la operación.

#### Lema 21.11 El

costo amortizado de cada operación MAKE-SET es  $O.1/$ .

Prueba Suponga que la operación  $q$ -ésima es  $\text{MAKE-SET.x/}$ . Esta operación crea el nodo  $x$  con rango 0, de modo que  $\text{qx/ D 0}$ . Ningún otro rango o potencial cambia, y por lo tanto  $\text{^q D } ^q1$ . Observando que el costo real de la operación MAKE-SET es  $O.1/$  completa la prueba. ■

#### Lema 21.12 El

costo amortizado de cada operación de LINK es  $O_{\cdot,n//}$ .

Prueba Suponga que la operación  $q$ -ésima es  $\text{ENLACE.x; y/}$ . El costo real de la operación LINK es  $O.1/$ . Sin pérdida de generalidad, suponga que LINK hace que  $y$  sea el parente de  $x$ .

Para determinar el cambio de potencial debido al ENLACE, observamos que los únicos nodos cuyos potenciales pueden cambiar son  $x$ ,  $y$  y los hijos de  $y$  justo antes de la operación. Mostraremos que el único nodo cuyo potencial puede aumentar debido al ENLACE es  $y$ , y que su aumento es como máximo  $_{\cdot,n/}$ :

Por el Lema 21.10, cualquier nodo que sea hijo de  $y$  justo antes del ENLACE no puede aumentar su potencial debido al ENLACE.

De la definición de  $\text{qx/}$ , vemos que, dado que  $x$  era una raíz justo antes de la operación  $q$ -ésima,  $\text{q1.x/ D } _{\cdot,n/x:rango}$ . Si  $x:rango D 0$ , entonces  $\text{qx/ D q1.x/ D 0}$ .

De lo contrario,

$\text{qx/ } < _{\cdot,n/} \text{x:rango}$  (por el Corolario 21.9)

$D q1.x/ ;$  y así

el potencial de  $x$  disminuye.

Como  $y$  es una raíz anterior a LINK,  $q_1.y / D \cup_n y : rank$ . La operación LINK deja  $y$  como raíz, y deja el rango de  $y$  solo o aumenta el rango de  $y$  en 1. Por lo tanto,  $qy / D q_1.y / o qy / D q_1.y / C \cup_n$ .

El aumento de potencial debido a la operación LINK , por lo tanto, es como máximo  $\cup_n$ .  
El costo amortizado de la operación LINK es  $O(1/C \cup_n D O \cup_n)$ . ■

Lema 21.13 El

costo amortizado de cada operación FIND-SET es  $O(\cup_n)$ .

Prueba Suponga que la operación  $q$ -ésima es FIND-SET y que la ruta de búsqueda contiene  $s$  nodos. El costo real de la operación FIND-SET es  $O(s)$ . Mostraremos que el potencial de ningún nodo aumenta debido al FIND-SET y que al menos  $\max(0; s \cup_n C 2) /$  los nodos en la ruta de búsqueda tienen su disminución potencial en al menos 1.

Para ver que el potencial de ningún nodo aumenta, primero apelamos al Lema 21.10 para todos los nodos excepto la raíz. Si  $x$  es la raíz, entonces su potencial es  $\cup_n x : rank$ , que no cambia.

Ahora mostramos que al menos  $\max(0; s \cup_n C 2) /$  los nodos tienen su disminución potencial en al menos 1. Sea  $x$  un nodo en el camino de búsqueda tal que  $x : rank > 0$  y  $x$  es seguido en algún lugar del camino de búsqueda por otro nodo  $y$  que no es una raíz, donde  $level.y / D level.x /$  justo antes de la operación FIND-SET . (No es necesario que el nodo  $y$  siga inmediatamente a  $x$  en la ruta de búsqueda.) Todos los nodos excepto como máximo  $\cup_n C 2$  en la ruta de búsqueda satisfacen estas restricciones en  $x$ . Los que no los satisfacen son el primer nodo de la ruta de búsqueda (si tiene rango 0), el último nodo de la ruta (es decir, la raíz) y el último nodo  $w$  de la ruta para qué nivel  $w / D k$ , para cada  $k D 0; 1; 2; \dots; \cup_n 1$ .

Fijemos tal nodo  $x$ , y mostraremos que el potencial de  $x$  disminuye en al menos 1. Sea  $k D level.x / D level.y /$ . Justo antes de la compresión de ruta causada por FIND-SET, tenemos  $A.iter.x // .x : rank /$  (por

$x : p : rango$	$\underset{k}{\text{definición de iter.}x /},$ (por definición de $level.y /$ ), (por
$y : p : rango$	$Ak.y : rango /$ corolario 21.5 y porque
$y : rango$	$x : p : rango$

y sigue a  $x$  en la ruta de búsqueda).

Juntando estas desigualdades y dejando que  $i$  sea el valor de  $iter.x /$  antes de la compresión de la ruta, tenemos

$y : p : rango$	$Ak.y : rank /$
	$Ak.x : p : rank /$ (porque $Ak.j /$ es estrictamente creciente)
	$Ak.A.iter.x // .x : rango //$
$D A.i C 1 / .x : rango / :$	

Debido a que la compresión de rutas hará que  $x$  e y tengan el mismo padre, sabemos que después de la compresión de rutas,  $x:p:rank$  D y  $y:p:rank$  y que la compresión de rutas no disminuye  $y:p:rank$ . Dado que  $x:rank$  no cambia, después de la compresión de ruta tenemos  $x:p:rank A.iC1/ .x:rank/$ . Por lo tanto, la compresión de ruta hará que  $\text{iter}.x/$  aumente (al menos a  $i C 1$ ) o  $\text{level}.x/$  (lo que ocurre si  $\text{iter}.x/$  aumenta al menos a  $x:rank C 1$ ). En cualquier caso, por el Lema 21.10, tenemos  $qx/ q1.x/ 1$ . Por lo tanto, el potencial de  $x$  disminuye en al menos 1.

El costo amortizado de la operación FIND-SET es el costo real más el cambio en el potencial. El costo real es  $O(s/)$ , y hemos demostrado que el potencial total disminuye al menos en un máximo de  $0; s \dots n/ C 2//$ . El coste amortizado, por tanto, es como máximo  $O(s \dots n/ C 2// D O(s/ s C O \dots n// D O \dots n//)$ , ya que podemos escalar las unidades de potencial para dominar la constante escondida en  $O(s/)$ . ■

Juntando los lemas anteriores se obtiene el siguiente teorema.

#### Teorema 21.14

Una secuencia de  $m$  operaciones MAKE-SET, UNION y FIND-SET,  $n$  de las cuales son operaciones MAKE-SET, se puede realizar en un bosque de conjuntos disjuntos con unión por rango y compresión de rutas en el peor de los casos  $O(m \cdot \text{norte} //)$ .

Prueba inmediata de los lemas 21.7, 21.11, 21.12 y 21.13. ■

#### Ejercicios

##### 21.4-1

Demostrar Lema 21.4.

##### 21.4-2

Demuestre que cada nodo tiene rango a lo sumo  $\lg n$ .

##### 21.4-3

A la luz del ejercicio 21.4-2, ¿cuántos bits son necesarios para almacenar  $x:rank$  para cada nodo  $x$ ?

##### 21.4-4

Utilizando el ejercicio 21.4-2, dé una demostración simple de que las operaciones en un bosque de conjuntos disjuntos con unión por rango pero sin compresión de trayectoria se ejecutan en tiempo  $O(m \lg n)$ .

##### 21.4-5

El profesor Dante razona que debido a que los rangos de los nodos aumentan estrictamente a lo largo de un camino simple hacia la raíz, los niveles de los nodos deben aumentar monótonamente a lo largo del camino. En otra

Es decir, si  $x:\text{rank} > 0$  y  $x:p$  no es una raíz, entonces  $\text{level}.x/\text{profesor}$  nivel. $x:p/$ . Es el correcto?

21.4-6 ?

Considere la función  $\text{O} .n/ \leq \min f_k W A_k / \lg n C 1/g$ . Muestre que  $\text{O} .n/$  para todos los valores prácticos de  $n$  y, usando el ejercicio 21.4-2, muestre cómo modificar el argumento de la función potencial para probar que podemos realizar una secuencia de  $m$  operaciones MAKE SET, UNION y FIND-SET ,  $n$  de las cuales son operaciones MAKE-SET , en un bosque de conjunto disjunto con unión por rango y compresión de ruta en el peor de los casos  $\text{O} .n//$ .

3

## Problemas

21-1 Mínimo fuera de línea El

problema de mínimo fuera de línea nos pide mantener un conjunto dinámico  $T$  de elementos del dominio  $f_1; 2; \dots; n$  bajo las operaciones INSERT y EXTRACT-MIN.

Nos dan una secuencia  $S$  de  $n$  llamadas INSERT y EXTRACT-MIN , donde cada tecla en  $f_1; 2; \dots; n$  se inserta exactamente una vez. Deseamos determinar qué clave devuelve cada llamada EXTRACT-MIN . Específicamente, deseamos llenar una matriz extraída  $E_i : : m$ , donde para  $i \in D$ ,  $E_i : : m$ ,  $E_i$  es la clave devuelta por la  $i$ -ésima llamada EXTRACT-MIN . El problema es "fuera de línea" en el sentido de que se nos permite procesar toda la secuencia  $S$  antes de determinar cualquiera de las claves devueltas.

a. En el siguiente caso del problema del mínimo fuera de línea, cada operación  $\text{INSERT}.i /$  está representada por el valor de  $i$  y cada  $\text{EXTRACT-MIN}$  está representada por la letra  $E$ :

$4; 8; MI; 3; MI; 9; 2; 6; MI; MI; MI; 1; 7; E; 5;$

Complete los valores correctos en la matriz extraída .

Para desarrollar un algoritmo para este problema, dividimos la secuencia  $S$  en subsecuencias homogéneas. Es decir, representamos  $S$  por

$I_1; E; I_2; E; I_3; \dots; I_m; E; I_m C_1 ;$

donde cada  $E$  representa una única llamada EXTRACT-MIN y cada  $I_j$  representa una secuencia (posiblemente vacía) de llamadas INSERT . Para cada subsecuencia  $I_j$  , colocamos inicialmente las claves insertadas por estas operaciones en un conjunto  $K_j$  , que está vacío si  $I_j$  está vacío.

Entonces hacemos lo siguiente:

FUERA DE LINEA-MINIMO.m; n/ 1

para i D 1 a n 2 3 4 5

determine j tal que i 2 Kj si j  $\leq$  m C 1

extraído D i

sea l el valor más

pequeño mayor que j para el cual existe el conjunto

Kl 6

KI D Kj [ Kl, destruyendo Kj 7 retorno

extraído

b. Argumente que la matriz extraída devuelta por OFF-LINE-MINIMUM es correcta.

C. Describir cómo implementar OFF-LINE-MINIMUM de manera eficiente con una estructura de datos de conjuntos disjuntos. Dé un límite estricto al tiempo de ejecución en el peor de los casos de su implementación.

#### 21-2 Determinación de la profundidad

En el problema de determinación de profundidad, mantenemos un bosque FD fTi g de árboles enraizados bajo tres operaciones:

MAKE-TREE./ crea un árbol cuyo único nodo es FIND-DEPTH./

devuelve la profundidad del nodo dentro de su árbol.

INJERTO.r; / hace que el nodo r, que se supone que es la raíz de un árbol, se convierta en el que se supone el hijo del nodo , que está en un árbol diferente al de r pero puede o puede no es en sí mismo una raíz.

a. Supongamos que usamos una representación de árbol similar a un bosque de conjunto disjunto: :p es el padre del nodo, excepto que :p D si es una raíz. Supongamos además que implementamos GRAFT.r; / configurando r:p D y FIND-DEPTH./ siguiendo la ruta de búsqueda hasta la raíz, devolviendo un recuento de todos los nodos que no sean los encontrados. Demuestre que el tiempo de ejecución en el peor de los casos de una secuencia de m operaciones CREAR ÁRBOL, ENCONTRAR PROFUNDIDAD e INJERTAR es ,.m2/.

Al usar las heurísticas de unión por rango y compresión de ruta, podemos reducir el tiempo de ejecución del peor de los casos. Usamos el bosque de conjuntos disjuntos SD fSi g, donde cada conjunto Si (que es en sí mismo un árbol) corresponde a un árbol Ti en el bosque F. La estructura de árbol dentro de un conjunto Si , sin embargo, no corresponde necesariamente a la de Ti .

De hecho, la implementación de Si no registra las relaciones padre-hijo exactas pero, sin embargo, nos permite determinar la profundidad de cualquier nodo en Ti .

La idea clave es mantener en cada nodo una “pseudodistancia” :d, que se define de modo que la suma de las pseudodistancias a lo largo del camino simple desde al

raíz de su conjunto Si es igual a la profundidad de en Ti . Es decir, si el camino simple desde su raíz en Si es D y es ~~la raíz de S~~ entonces la profundidad de en Ti es Pk

jD0 j

b. Dé una implementación de MAKE-TREE.

C. Muestre cómo modificar FIND-SET para implementar FIND-DEPTH. Su implementación debe realizar una compresión de ruta y su tiempo de ejecución debe ser lineal en la longitud de la ruta de búsqueda. Asegúrese de que su implementación actualice las pseudodistancias correctamente.

d. Mostrar cómo implementar GRAFT.r; / , que combina los conjuntos que contienen r y , modificando los procedimientos UNION y LINK . Asegúrese de que su implementación actualice las pseudodistancias correctamente. Tenga en cuenta que la raíz de un conjunto Si no es necesariamente la raíz del árbol Ti correspondiente .

mi. Proporcione un límite estrecho para el tiempo de ejecución en el peor de los casos de una secuencia de m operaciones CREAR ÁRBOL, ENCONTRAR PROFUNDIDAD e INJERTAR , n de las cuales son operaciones CREAR ÁRBOL .

21-3 Algoritmo de ancestros menos comunes fuera de línea de Tarjan El ancestro menos común de dos nodos u y en un árbol con raíz T es el nodo w que es un ancestro de u y y que tiene la mayor profundidad en T En el off -problema de . antecesores menos comunes de línea, se nos da un árbol enraizado T y un conjunto arbitrario PD ffu; gg de pares desordenados de nodos en T mina , y deseamos disuadir el ancestro menos común de cada par en P.

Para resolver el problema de los ancestros menos comunes fuera de línea, el siguiente procedimiento realiza un recorrido por el árbol de T con la llamada inicial LCA.T:root/. Suponemos que cada nodo está coloreado de BLANCO antes de la caminata.

LCA.u/ 1

MAKE-SET.u/ 2

FIND-SET.u/:ancestro D u 3 para  
cada hijo de u en T 4 LCA./ 5

UNION.u; / 6 FIND-

SET.u/:ancestor D u 7

u:color D BLACK 8 para cada nodo tal  
que fu; g 2 P 9

si :color == NEGRO

10               escribe “El ancestro menos común de”  
                  u “y” “es” FIND-SET./:ancestro

- a. Argumente que la línea 10 se ejecuta exactamente una vez para cada par  $f_u; g$  2 p.
- b. Argumente que en el momento de la llamada LCA.u/, el número de conjuntos en el conjunto disjunto estructura de datos es igual a la profundidad de  $u$  en  $T$ .
- C. Demuestre que LCA imprime correctamente el antepasado menos común de  $u$  y para cada par  $f_u; g$  2 p.
- d. Analice el tiempo de ejecución de LCA, asumiendo que usamos la implementación de la estructura de datos de conjuntos disjuntos de la Sección 21.3.

### Notas del capítulo

Muchos de los resultados importantes para las estructuras de datos de conjuntos disjuntos se deben, al menos en parte, a RE Tarjan. Usando el análisis agregado, Tarjan [328, 330] dio el primer límite superior ajustado en términos de la inversa de crecimiento muy lento  $\sim m^y n^y$  de la función de Ackermann. (La función  $Ak.j$  / dada en la Sección 21.4 es similar a la función de Ackermann, y la función  $\sim n^y$  es similar a la inversa. Tanto  $\sim n^y$  como  $\sim m^y n^y$  son como máximo 4 para todos los valores concebibles de  $m$  y  $n$ .) Hopcroft y Ullman [5, 179] probaron antes un límite superior de  $O(m \lg n)$ . El tratamiento de la Sección 21.4 está adaptado de un análisis posterior de Tarjan [332], que a su vez se basa en un análisis de Kozen [220]. Harfst y Reingold [161] dan una versión basada en el potencial del límite anterior de Tarjan.

Tarjan y van Leeuwen [333] analizan variantes de la heurística de compresión de rutas, incluidos los "métodos de un paso", que a veces ofrecen mejores factores constantes en su rendimiento que los métodos de dos pasos. Al igual que con los análisis anteriores de Tarjan de la heurística básica de compresión de rutas, los análisis de Tarjan y van Leeuwen son agregados. Harfst y Reingold [161] mostraron más tarde cómo hacer un pequeño cambio en la función potencial para adaptar su análisis de compresión de trayectoria a estas variantes de un solo paso. Gabow y Tarjan [121] muestran que en ciertas aplicaciones, las operaciones de conjuntos disjuntos pueden ejecutarse en tiempo  $O(m)$ .

Tarjan [329] mostró que un límite inferior de  $\sim m^y n^y$  se requiere tiempo para operaciones en cualquier estructura de datos de conjunto disjunto que satisfaga ciertas condiciones técnicas. Este límite inferior fue posteriormente generalizado por Fredman y Saks [113], quienes demostraron que en el peor de los casos,  $\sim m^y n^y$ . Se debe acceder a palabras de memoria de  $\lg n$ -bit.

---

## Algoritmos de gráfico VI

---

## Introducción

Los problemas de grafos impregnan las ciencias de la computación, y los algoritmos para trabajar con ellos son fundamentales para el campo. Cientos de problemas computacionales interesantes se expresan en términos de gráficos. En esta parte, mencionamos algunos de los más significativos.

El Capítulo 22 muestra cómo podemos representar un gráfico en una computadora y luego analiza los algoritmos basados en la búsqueda de un gráfico utilizando la búsqueda primero en amplitud o la búsqueda primero en profundidad. El capítulo ofrece dos aplicaciones de la búsqueda primero en profundidad: clasificar topológicamente un gráfico acíclico dirigido y descomponer un gráfico dirigido en sus componentes fuertemente conectados.

El Capítulo 23 describe cómo calcular un árbol de expansión de peso mínimo de un gráfico: la forma de peso mínimo de conectar todos los vértices cuando cada borde tiene un peso asociado. Los algoritmos para calcular árboles de expansión mínimos sirven como buenos ejemplos de algoritmos codiciosos (consulte el Capítulo 16).

Los capítulos 24 y 25 consideran cómo calcular los caminos más cortos entre vértices cuando cada borde tiene una longitud o "peso" asociado. El Capítulo 24 muestra cómo encontrar los caminos más cortos desde un vértice de origen dado a todos los demás vértices, y el Capítulo 25 examina los métodos para calcular los caminos más cortos entre cada par de vértices.

Finalmente, el Capítulo 26 muestra cómo calcular un flujo máximo de material en una red de flujo, que es un gráfico dirigido que tiene un vértice fuente de material específico, un vértice receptor específico y capacidades específicas para la cantidad de material que puede atravesar cada borde dirigido. . Este problema general surge de muchas formas, y un buen algoritmo para calcular los flujos máximos puede ayudar a resolver eficientemente una variedad de problemas relacionados.

Cuando caracterizamos el tiempo de ejecución de un algoritmo gráfico en un gráfico dado  $G$ ;  $.V; E/$ , normalmente medimos el tamaño de la entrada en términos del número de vértices  $|V|$  y el número de aristas  $|E|$  del gráfico. Es decir, describimos el tamaño de la entrada con dos parámetros, no solo con uno. Adoptamos una convención notacional común para estos parámetros. Dentro de la notación asintótica (como la notación  $O$  o la notación  $\sim$ ), y solo dentro de dicha notación, el símbolo  $V$  denota  $|V|$  y el símbolo  $E$  denota  $|E|$ . Por ejemplo, podríamos decir, "el algoritmo se ejecuta en el tiempo  $O.VE/$ ", lo que significa que el algoritmo se ejecuta en el tiempo  $O(|V| |E|)$ . Esta convención hace que las fórmulas de tiempo de ejecución sean más fáciles de leer, sin riesgo de ambigüedad.

Otra convención que adoptamos aparece en pseudocódigo. Denotamos el conjunto de vértices de un grafo  $G$  por  $G.V$  y su conjunto de aristas por  $G.E$ . Es decir, el pseudocódigo ve los conjuntos de vértices y aristas como atributos de un gráfico.

---

## 22

# Algoritmos de grafos elementales

Este capítulo presenta métodos para representar un gráfico y para buscar un gráfico.

Buscar un gráfico significa seguir sistemáticamente los bordes del gráfico para visitar los vértices del gráfico. Un algoritmo de búsqueda de gráficos puede descubrir mucho sobre la estructura de un gráfico. Muchos algoritmos comienzan buscando en su gráfico de entrada para obtener esta información estructural. Varios otros algoritmos de gráficos elaboran la búsqueda básica de gráficos. Las técnicas para buscar un gráfico se encuentran en el corazón del campo de los algoritmos de gráficos.

La Sección 22.1 analiza las dos representaciones computacionales más comunes de grafos: como listas de adyacencia y como matrices de adyacencia. La sección 22.2 presenta un algoritmo simple de búsqueda de gráficos llamado búsqueda primero en anchura y muestra cómo crear un árbol primero en anchura. La Sección 22.3 presenta la búsqueda primero en profundidad y prueba algunos resultados estándar sobre el orden en que la búsqueda primero en profundidad visita los vértices. La sección 22.4 proporciona nuestra primera aplicación real de la búsqueda primero en profundidad: ordenar topológicamente un gráfico acíclico dirigido. Una segunda aplicación de la búsqueda primero en profundidad, encontrar los componentes fuertemente conectados de un grafo dirigido, es el tema de la Sección 22.5.

---

## 22.1 Representaciones de gráficos

Podemos elegir entre dos formas estándar de representar un gráfico  $G$ :  $V; E$ : como colección de listas de adyacencia o como matriz de adyacencia. Cualquiera de las dos formas se aplica tanto a grafos dirigidos como no dirigidos. Debido a que la representación de lista de adyacencia proporciona una forma compacta de representar gráficos dispersos, aquellos para los que  $|E|$  es mucho menor que  $|V|$  presentados en este libro se supone que un  $2^{|V|}$  —suele ser el método de elección. La mayoría de los algoritmos de grafos de entrada se representa en forma de lista de adyacencia. Sin embargo, podemos preferir una representación de matriz de adyacencia cuando el gráfico es denso:  $|E|$  está cerca de  $|V|^2$  si hay una arista que conecta dos vértices dados. Por ejemplo, dos  $2^{|V|}$  —o cuando necesitamos ser capaces de decirlo rápidamente de los todos los pares

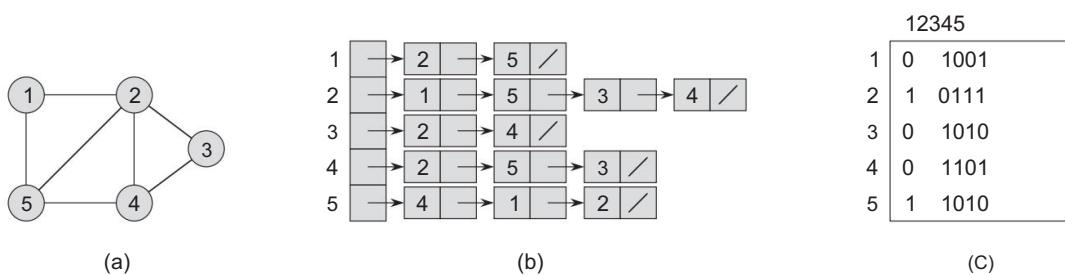


Figura 22.1 Dos representaciones de un grafo no dirigido. (a) Un grafo no dirigido  $G$  con 5 vértices y 7 aristas. (b) Una representación de lista de adyacencia de  $G$ . (c) La representación de matriz de adyacencia de  $G$ .

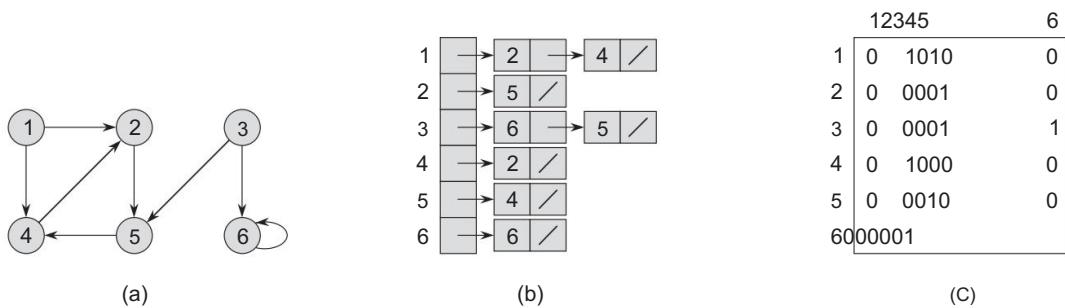


Figura 22.2 Dos representaciones de un gráfico dirigido. (a) Un gráfico dirigido  $G$  con 6 vértices y 8 aristas. (b) Una representación de lista de adyacencia de  $G$ . (c) La representación de matriz de adyacencia de  $G$ .

Los algoritmos de caminos más cortos presentados en el Capítulo 25 asumen que sus gráficos de entrada están representados por matrices de adyacencia.

La representación de lista de adyacencia de un grafo  $G$ ;  $V$ ;  $E$  consta de un arreglo  $Adj$  de  $|V|$  listas, una para cada vértice en  $V$ . Para cada  $u \in V$  la lista de adyacencia  $Adj[u]$  contiene todos los vértices tales que hay una arista  $u; v \in E$ . Es decir,  $Adj[u]$  consta de todos los vértices adyacentes a  $u$  en  $G$ . (Alternativamente, puede contener punteros a estos vértices). Dado que las listas de adyacencia representan los bordes de un gráfico, en pseudocódigo tratamos la matriz  $Adj$  como un atributo del grafo, tal como tratamos el conjunto de aristas  $E$ . En pseudocódigo, por lo tanto, veremos una notación como  $G:Adj$ .

La figura 22.1(b) es una representación de lista de adyacencia del grafo no dirigido de la figura 22.1(a). De manera similar, la figura 22.2(b) es una representación de lista de adyacencia del gráfico dirigido de la figura 22.2(a).

Si  $G$  es un gráfico dirigido, la suma de las longitudes de todas las listas de adyacencia es  $|E|$ , ya que una arista de la forma  $u; v$  se representa por haber aparecido en  $Adj[u]$  si  $v$  es

un grafo no dirigido, la suma de las longitudes de todas las listas de adyacencia es  $2|E|$ , ya que si  $.u; /$  es un borde no dirigido, entonces  $u$  aparece en la lista de adyacencia de  $v$  y viceversa.

Tanto para gráficos dirigidos como no dirigidos, la representación de lista de adyacencia tiene la propiedad deseable de que la cantidad de memoria que requiere es  $.VCE|$ .

Podemos adaptar fácilmente las listas de adyacencia para representar gráficas ponderadas, es decir, gráficas en las que cada borde tiene un peso asociado, típicamente dado por una función de peso  $w : E \rightarrow \mathbb{R}$ . Por ejemplo, sea  $GD = (V, E)$  sea un gráfico ponderado con función de peso  $w$ . Simplemente almacenamos el peso  $w_{uv}$  del borde  $.u; / v$  con vértice  $u$  en la lista de adyacencia de  $v$ . La representación de la lista de adyacencia es bastante sólida, ya que podemos modificarla para admitir muchas otras variantes de gráficos.

Una desventaja potencial de la representación de lista de adyacencia es que no proporciona una forma más rápida de determinar si un borde dado  $.u; /$  está presente en el gráfico que buscar en la lista de adyacencia  $Adj(G)$ . Una representación de matriz de adyacencia del gráfico soluciona esta desventaja, pero a costa de usar asintóticamente más memoria. (Consulte el Ejercicio 22.1-8 para obtener sugerencias de variaciones en las listas de adyacencia que permiten una búsqueda de bordes más rápida).

Para la representación matricial de adyacencia de un grafo  $GD = (V, E)$ , suponemos que los vértices están numerados  $1, 2, \dots, |V|$  de alguna manera arbitraria. Entonces la representación matricial de adyacencia de un grafo  $G$  consiste en una matriz  $AD = [a_{ij}] \in \mathbb{R}^{V \times V}$  tal que

$$a_{ij} = \begin{cases} 1 & \text{si } .i; / j \text{ es un borde} \\ 0 & \text{de lo contrario:} \end{cases}$$

Las figuras 22.1(c) y 22.2(c) son las matrices de adyacencia de los gráficos dirigidos y no dirigidos de las figuras 22.1(a) y 22.2(a), respectivamente. La matriz de adyacencia de un gráfico requiere  $.V^2$  de memoria, independientemente del número de aristas en el gráfico.

Observe la simetría a lo largo de la diagonal principal de la matriz de adyacencia en la figura 22.1(c). Ya que en un grafo no dirigido,  $.u; / v$  representan la misma arista, la matriz de adyacencia  $A$  de un grafo no dirigido es su propia traspuesta:  $AD = AT$ .

En algunas aplicaciones, vale la pena almacenar solo las entradas en y por encima de la diagonal de la matriz de adyacencia, reduciendo así la memoria necesaria para almacenar el gráfico casi a la mitad.

Al igual que la representación de lista de adyacencia de un gráfico, una matriz de adyacencia puede representar un gráfico ponderado. Por ejemplo, si  $GD = (V, E)$  es un gráfico ponderado con una función de peso de borde  $w$ , simplemente podemos almacenar el peso  $w_{uv}$  del borde  $.u; / v$  como la entrada en la fila  $u$  y la columna de la matriz de adyacencia. Si no existe un borde, podemos almacenar un valor NIL como su entrada de matriz correspondiente, aunque para muchos problemas es conveniente usar un valor como 0 o 1.

Aunque la representación de la lista de adyacencia es asintóticamente al menos tan eficiente en cuanto al espacio como la representación de la matriz de adyacencia, las matrices de adyacencia son más simples y, por lo tanto, es posible que las prefiramos cuando los gráficos son razonablemente pequeños. Además, adja-

Las matrices de cencia tienen una ventaja adicional para los gráficos no ponderados: solo requieren un bit por entrada.

### Representando atributos

La mayoría de los algoritmos que operan en gráficos necesitan mantener atributos para vértices y/o aristas. Indicamos estos atributos usando nuestra notación habitual, como :d para un atributo d de un vértice. Cuando indicamos los bordes como pares de vértices, usamos el mismo estilo de notación. Por ejemplo, si los bordes tienen un atributo f , indique este atributo para el borde .u; / por .u; //f . Entonces nosotros

Para el propósito de presentar y comprender los algoritmos, nuestra notación de atributos es suficiente.

La implementación de atributos de vértice y borde en programas reales puede ser otra historia completamente diferente. No existe una mejor manera de almacenar y acceder a los atributos de vértice y borde. Para una situación dada, su decisión probablemente dependerá del lenguaje de programación que esté usando, el algoritmo que esté implementando y cómo el resto de su programa use el gráfico. Si representa un gráfico usando listas de adyacencia, un diseño representa atributos de vértice en matrices adicionales, como una matriz  $d \in \mathbb{C}^{V \times V}$  que es paralela a la matriz  $Adj$  . Si los vértices adyacentes a u están en  $Adj[u]$ , entonces lo que llamamos el atributo  $u:d$  en realidad se almacenaría en la entrada del arreglo  $d[u]$ . Son posibles muchas otras formas de implementar atributos. Por ejemplo, en un lenguaje de programación orientado a objetos, los atributos de vértice pueden representarse como variables de instancia dentro de una subclase de una clase de vértice.

## Ejercicios

### 22.1-1

Dada una representación de lista de adyacencia de un gráfico dirigido, ¿cuánto tiempo lleva calcular el grado de salida de cada vértice? ¿Cuánto tiempo se tarda en calcular los grados?

### 22.1-2

Dé una representación de lista de adyacencia para un árbol binario completo en 7 vértices. Dé una representación de matriz de adyacencia equivalente. Suponga que los vértices están numerados del 1 al 7 como en un montón binario.

### 22.1-3

La transpuesta de un grafo dirigido  $G = (V, E)$  es el grafo  $G^T = (V, E^T)$ , donde  $E^T = \{(u, v) \mid (v, u) \in E\}$ . Así,  $G^T$  es  $G$  con todas sus aristas invertidas.

Describa algoritmos eficientes para calcular  $G^T$  a partir de  $G$ , tanto para la lista de adyacencia como para las representaciones de matriz de adyacencia de  $G$ . Analice los tiempos de ejecución de sus algoritmos.

## 22.1-4

Dada una representación de lista de adyacencia de un multigrafo  $G = \langle V; E \rangle$ , describe un algoritmo  $O(V|E|)$ -tiempo para calcular la representación de lista de adyacencia del grafo no dirigido "equivalente"  $G' = \langle V; E' \rangle$ , donde  $E'$  consta de las aristas en  $E$  con todas las aristas múltiples entre dos vértices reemplazadas por una sola arista y con todos los bucles propios eliminados.

## 22.1-5

El cuadrado de un grafo dirigido  $G = \langle V; E \rangle$  es el grafo  $G^2 = \langle V; E^2 \rangle$  tal que  $u; v \in E^2$  si y solo si  $G$  contiene una ruta con como máximo dos aristas entre  $u$  y  $v$ . Describa algoritmos eficientes para calcular  $G^2$  a partir de  $G$  tanto para la lista de adyacencia como para las representaciones de matriz de adyacencia de  $G$ . Analice los tiempos de ejecución de sus algoritmos.

## 22.1-6

La mayoría de los algoritmos de grafos que toman una representación de matriz de adyacencia como entrada requieren tiempo  $O(V^2)$ , pero hay algunas excepciones. Muestre cómo determinar si un grafo dirigido  $G$  contiene un sumidero universal (un vértice con grado de entrada  $j_V > 1$  y grado de salida 0) en el tiempo  $O(V)$ , dada una matriz de adyacencia para  $G$ .

## 22.1-7

La matriz de incidencia de un grafo dirigido  $G = \langle V; E \rangle$  sin bucles propios es una matriz  $|V| \times |E|$   $B = [b_{ij}]$  tal que

$$\begin{aligned} b_{ij} &= 1 && \text{si la arista } j \text{ sale del vértice } i \text{ si } \\ && & : \\ b_{ij} &= 1 && \text{la arista } j \text{ entra en el vértice } i \text{ si } \\ && & : \\ &0 && \text{de lo contrario:} \end{aligned}$$

Describe qué representan las entradas del producto de matriz  $B^T B$ , donde  $B^T$  es la transpuesta de  $B$ .

## 22.1-8

Suponga que en lugar de una lista enlazada, cada entrada de matriz  $Adj(E)$  es una tabla hash que contiene los vértices para los cuales  $u; v \in E$ . Si todas las búsquedas de aristas son igualmente probables, ¿cuál es el tiempo esperado para determinar si una arista está en el gráfico? ¿Qué desventajas tiene este esquema? Sugiera una estructura de datos alternativa para cada lista de aristas que resuelva estos problemas. ¿Tu alternativa tiene desventajas en comparación con la tabla hash?

## 22.2 Búsqueda en amplitud

La búsqueda en amplitud es uno de los algoritmos más simples para buscar un gráfico y el arquetipo de muchos algoritmos de gráficos importantes. El algoritmo de árbol de expansión mínima de Prim (Sección 23.2) y el algoritmo de caminos más cortos de fuente única de Dijkstra (Sección 24.3) utilizan ideas similares a las de la búsqueda en amplitud.

Dado un grafo  $G = (V; E)$  y un vértice fuente distinguido  $s$ , la búsqueda primero en amplitud explora sistemáticamente los bordes de  $G$  para "descubrir" todos los vértices que son accesibles desde  $s$ . Calcula la distancia (menor número de aristas) desde  $s$  hasta cada vértice alcanzable. También produce un "árbol primero en anchura" con raíz  $s$  que contiene todos los vértices alcanzables. Para cualquier vértice alcanzable desde  $s$ , el camino simple en el árbol primero en anchura desde  $s$  hasta corresponde a un "camino más corto" desde  $s$  hasta en  $G$ , es decir, un camino que contiene el menor número de aristas. El algoritmo funciona tanto en gráficos dirigidos como no dirigidos.

La búsqueda primero en amplitud se denomina así porque expande la frontera entre los vértices descubiertos y no descubiertos de manera uniforme a lo largo de la anchura de la frontera. Es decir, el algoritmo descubre todos los vértices a la distancia  $k$  de  $s$  antes de descubrir cualquier vértice a la distancia  $k + 1$ .

Para realizar un seguimiento del progreso, la búsqueda en anchura colorea cada vértice de blanco, gris o negro. Todos los vértices comienzan siendo blancos y luego pueden volverse grises y luego negros. Un vértice se descubre la primera vez que se encuentra durante la búsqueda, momento en el cual se vuelve no blanco. Por lo tanto, se han descubierto vértices grises y negros, pero la búsqueda primero en anchura distingue entre ellos para garantizar que la búsqueda procede de una manera primero en anchura.<sup>1</sup> Si  $u$  es el vértice  $u$  es negro, entonces el vértice  $u$  es gris o negro; es decir, se han descubierto todos los vértices adyacentes a los vértices negros. Los vértices grises pueden tener algunos vértices blancos adyacentes; representan la frontera entre vértices descubiertos y no descubiertos.

La búsqueda primero en anchura construye un árbol primero en anchura, que inicialmente contiene solo su raíz, que es el vértice de origen  $s$ . Siempre que la búsqueda descubra un vértice blanco en el curso de la exploración de la lista de adyacencia de un vértice  $u$  ya descubierto, el vértice  $u$  y la arista  $(u, v)$  se agregan al árbol. Decimos que  $v$  es el predecesor o el parente de  $v$  en el árbol primero en anchura. Dado que un vértice se descubre como máximo una vez, tiene como máximo un parente. Las relaciones de ancestro y descendiente en el árbol primero en anchura se definen en relación con la raíz  $s$  como de costumbre: si  $v$  está en el camino simple en el árbol desde la raíz  $s$  hasta el vértice  $v$ , entonces  $v$  es un ancestro y un descendiente de  $v$ .

,

<sup>1</sup> Distinguimos entre vértices grises y negros para ayudarnos a entender cómo opera la búsqueda primero en amplitud. De hecho, como muestra el ejercicio 22.2-3, obtendríamos el mismo resultado incluso si no distinguiésemos entre vértices grises y negros.

El procedimiento de búsqueda primero en amplitud BFS a continuación asume que el gráfico de entrada  $G = \langle V, E \rangle$  se representa mediante listas de adyacencia. Adjunta varios atributos adicionales a cada vértice del gráfico. Almacenamos el color de cada vértice  $u \in V$  en el atributo  $u:\text{color}$  y el predecesor de  $u$  en el atributo  $u:\text{parent}$ . Si  $u$  no tiene predecesor (por ejemplo, si  $u$  es la fuente  $s$  o  $u$  no ha sido descubierto), entonces  $u:\text{parent} = \text{NIL}$ .

El atributo  $u:\text{dist}$  contiene la distancia desde la fuente  $s$  hasta el vértice  $u$  calculado por el algoritmo. El algoritmo también utiliza una cola  $Q$  de primero en entrar, primero en salir (consulte la Sección 10.1) para administrar el conjunto de vértices grises.

BFS.G;  $s: 1$

```

para cada vértice  $u \in V$ 
    fsg 2  $u:\text{color} = \text{BLANCO}$ 
     $u:\text{parent} = \text{NIL}$ 
     $u:\text{dist} = \infty$ 
     $u:\text{color} = \text{GRIS}$ 
     $u:\text{parent} = \text{NIL}$ 
     $u:\text{dist} = 0$ 
     $u:\text{parent} = s$ 
     $u:\text{dist} = 1$ 
     $u:\text{parent} = \text{NIL}$ 
    NIL

```

```

8  $Q = \emptyset$ ;  $9$ 
ENQUEUE.Q;  $s: 10$ 
mientras  $Q \neq \emptyset$  ;  $u: D$ 
11     DEQUEUE.Q/ por cada
12     2  $G:\text{Adj}[u]$  si  $:\text{color} == \text{BLANCO}$ 
13         :color  $D = \text{GRIS}$ 
14         :d  $tu:\text{dist} + 1$ 
15         :d  $tu:\text{parent} = u$ 
16         :d  $tu:\text{parent} = \text{NIL}$ 
17         ENQUEUE.Q; /
18      $u:\text{color} = \text{NEGRO}$ 

```

La Figura 22.3 ilustra el progreso de BFS en un gráfico de muestra.

El procedimiento BFS funciona de la siguiente manera. Con la excepción del vértice de origen  $s$ , las líneas 1 a 4 pintan todos los vértices de blanco, configuran  $u:\text{parent}$  para que sea infinito para cada vértice  $u$  y configuran el parent de cada vértice para que sea NIL. La línea 5 pinta  $s$  gris, ya que consideramos que se descubre cuando se inicia el procedimiento. La línea 6 inicializa  $s:\text{parent}$  a 0, y la línea 7 establece que el predecesor de la fuente sea NIL. Las líneas 8 y 9 inicializan  $Q$  en la cola que contiene solo el vértice  $s$ .

El ciclo while de las líneas 10 a 18 itera mientras queden vértices grises, que son vértices descubiertos cuyas listas de adyacencia aún no se han examinado por completo. Este ciclo while mantiene el siguiente invariante:

En la prueba de la línea 10, la cola  $Q$  consta del conjunto de vértices grises.

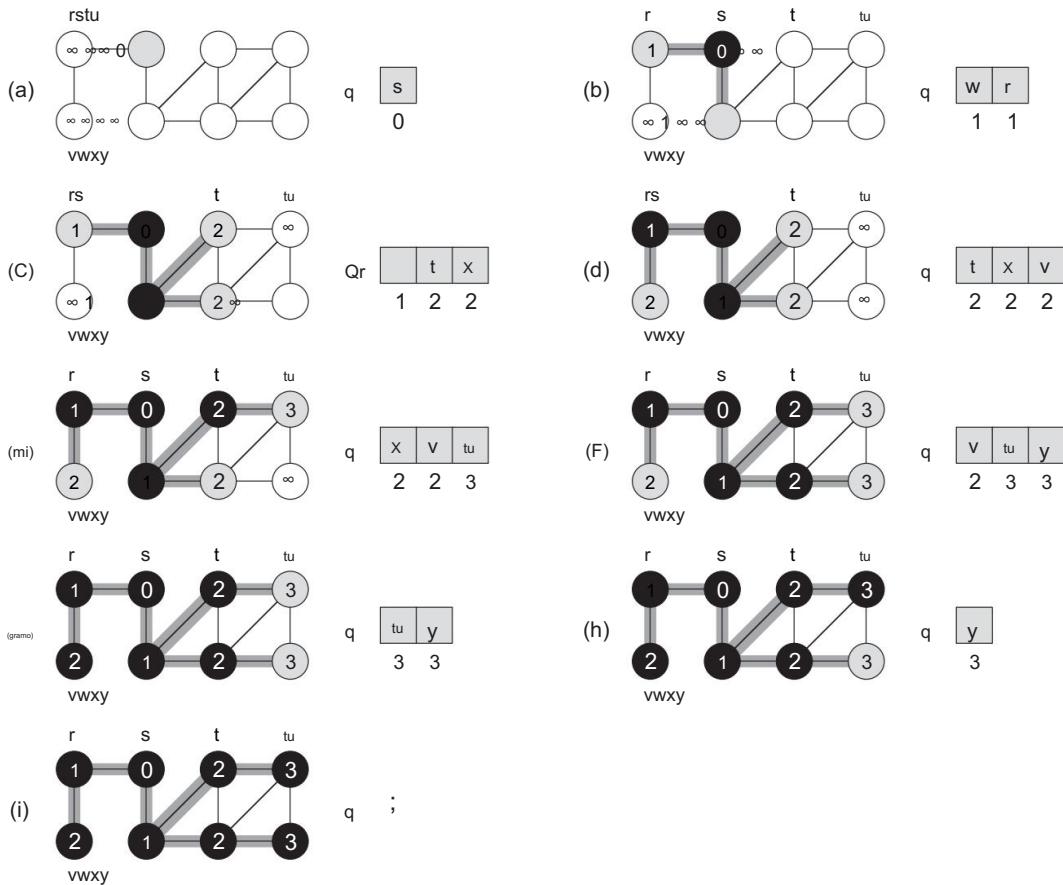


Figura 22.3 La operación de BFS en un gráfico no dirigido. Los bordes de los árboles se muestran sombreados, ya que los produce BFS. El valor de  $u:d$  aparece dentro de cada vértice  $u$ . La cola  $Q$  se muestra al comienzo de cada iteración del ciclo while de las líneas 10–18. Las distancias de los vértices aparecen debajo de los vértices en la cola.

Aunque no usaremos este ciclo invariante para probar la corrección, es fácil ver que se mantiene antes de la primera iteración y que cada iteración del ciclo mantiene el invariante. Antes de la primera iteración, el único vértice gris y el único vértice en  $Q$  es el vértice de origen  $s$ . La línea 11 determina el vértice gris  $u$  en la cabecera de la cola  $Q$  y lo elimina de  $Q$ . El ciclo for de las líneas 12 a 17 considera cada vértice en la lista de adyacencia de  $u$ . Si es blanco, aún no se ha descubierto y el procedimiento lo descubre ejecutando las líneas 14–17. El procedimiento pinta el vértice gris, establece su distancia  $:d$  a  $u:dC1$ , registra  $u$  como su parente  $:y$  y lo coloca al final de la cola  $Q$ . Una vez que el procedimiento ha examinado todos los vértices en  $u$

lista de adyacencia, ennegrece u en la línea 18. El ciclo invariante se mantiene porque cada vez que un vértice se pinta de gris (en la línea 14) también se pone en cola (en la línea 17), y cada vez que un vértice se quita de la cola (en la línea 11) se también pintado de negro (en la línea 18).

Los resultados de la búsqueda en anchura pueden depender del orden en que se visitan los vecinos de un vértice dado en la línea 12: el árbol en anchura puede variar, pero no así las distancias d calculadas por el algoritmo. (Consulte el ejercicio 22.2-5.)

### Análisis

Antes de probar las diversas propiedades de la búsqueda primero en amplitud, asumimos el trabajo más sencillo de analizar su tiempo de ejecución en un gráfico de entrada  $G = (V, E)$ . Usamos el análisis agregado, como vimos en la Sección 17.1. Después de la inicialización, la búsqueda primero en amplitud nunca blanquea un vértice y, por lo tanto, la prueba en la línea 13 asegura que cada vértice se pone en cola como máximo una vez y, por lo tanto, se elimina de la cola como máximo una vez. Las operaciones de encolado y desencolado toman  $O(1)$  de tiempo, por lo que el tiempo total dedicado a las operaciones en cola es de  $O(V)$ . Debido a que el procedimiento escanea la lista de adyacencia de cada vértice solo cuando el vértice está fuera de la cola, escanea cada lista de adyacencia como máximo una vez. Dado que la suma de las longitudes de todas las listas de adyacencia es  $|E|$ , el tiempo total empleado en escanear las listas de adyacencia es  $O(|E|)$ . La sobrecarga para la inicialización es  $O(V)$  y, por lo tanto, el tiempo total de ejecución del procedimiento BFS es  $O(V + |E|)$ . Por lo tanto, la búsqueda primero en amplitud se ejecuta en el tiempo de forma lineal en el tamaño de la representación de lista de adyacencia de  $G$ .

### Caminos más cortos

Al comienzo de esta sección, dijimos que la búsqueda primero en amplitud encuentra la distancia a cada vértice alcanzable en un gráfico  $G = (V, E)$  de un vértice fuente dado  $s \in V$ . Defina la distancia del camino más corto  $i(s; t)$  como el número mínimo de aristas en cualquier camino desde el vértice  $s$  hasta el vértice  $t$ ; si no hay camino de  $s$  a  $t$  entonces  $i(s; t) = \infty$ . Llamamos a un camino de longitud  $i(s; t)$  el camino más corto de  $s$  a  $t$ . Antes de mostrar que la búsqueda primero en amplitud calcula correctamente las distancias de ruta más cortas, investigamos una propiedad importante de las distancias de ruta más corta.

<sup>2</sup> En los capítulos 24 y 25, generalizaremos nuestro estudio de los caminos más cortos a gráficos ponderados, en los que cada borde tiene un peso de valor real y el peso de un camino es la suma de los pesos de sus bordes constituyentes. Los gráficos considerados en el presente capítulo son sin ponderar o, de manera equivalente, todas las aristas tienen peso unitario.

**Lema 22.1**

Sea  $G = (V, E)$ ; Sea  $E'$  un grafo dirigido o no dirigido, y sea  $s \in V$  un vértice arbitrario. Entonces, para cualquier arista  $(u, v) \in E'$ ,

$\text{es; } / \text{ es; tu/C 1}$

Prueba Si  $u$  es accesible desde  $s$ , entonces también lo es. En este caso, el camino más corto desde  $s$  hasta  $u$  no puede ser más largo que el camino más corto desde  $s$  hasta  $v$  seguido por la arista  $(u, v)$ , y por lo tanto se cumple la desigualdad. Si  $u$  no es accesible desde  $s$ , entonces  $\text{is; u/D 1}$ , y la desigualdad se cumple. ■

Queremos mostrar que BFS calcula correctamente  $\text{d}[v] = \text{is; } /$  para cada vértice  $v \in V$ . Primero mostramos que  $\text{d}[v]$  limita con  $\text{is; } /$  desde arriba.

**Lema 22.2**

Sea  $G = (V, E)$ ; Sea  $E'$  un grafo dirigido o no dirigido, y suponga que BFS se ejecuta en  $G$  desde un vértice fuente dado  $s \in V$ . Luego, al terminar, para cada vértice  $v \in V$ , el valor  $\text{d}[v]$  calculado por BFS satisface  $\text{d}[v] \leq \text{is; } /$ .

Prueba Usamos inducción sobre el número de operaciones ENQUEUE. Nuestra hipótesis inductiva es que  $\text{d}[v] \leq \text{is; } /$  para todos los  $v \in V$ .

La base de la inducción es la situación inmediatamente después de poner en cola  $s$  en la línea 9 de BFS. La hipótesis inductiva se cumple aquí, porque  $\text{d}[s] = 0 \leq \text{is; } /$  y  $\text{d}[v] = 1 \leq \text{is; } /$  para todos los  $v \in V$  fsg.

Para el paso inductivo, considere un vértice blanco que se descubre durante la búsqueda desde un vértice  $u$ . La hipótesis inductiva implica que  $\text{d}[v] \leq \text{is; } /$  para todos los vértices blancos  $v$ . De la asignación realizada por la línea 15 y del Lema 22.1, se obtiene

$\text{d}[v] = \text{d}[u] + 1$

$\text{es; } / \text{ C 1}$

$\text{is; } / :$

Vertex luego se pone en cola, y nunca se vuelve a poner en cola porque también está atenuado y la cláusula "then" de las líneas 14 a 17 se ejecuta solo para los vértices blancos. Por lo tanto, el valor de  $\text{d}[v]$  nunca vuelve a cambiar y la hipótesis inductiva se mantiene. ■

Para probar que  $\text{d}[v] \leq \text{is; } /$ , primero debemos mostrar con mayor precisión cómo opera la cola  $Q$  durante el curso de BFS. El siguiente lema muestra que en todo momento, la cola tiene como máximo dos valores distintos.

**Lema 22.3**

Suponga que durante la ejecución de BFS en un grafo  $G = (V, E)$ , la cola  $Q$  contiene los vértices  $h_1; h_2; \dots; h_i$ , donde  $h_i$  es la cabeza de la cola. Entonces  $d(h_1) \leq d(h_2) \leq \dots \leq d(h_i)$ .

Prueba La prueba es por inducción sobre el número de operaciones en cola. Inicialmente, cuando la cola contiene solo  $s$ , el lema ciertamente se cumple.

Para el paso inductivo, debemos demostrar que el lema se cumple después de quitar y poner en cola un vértice. Si se elimina la cabeza de la cola,  $h_i$  se convierte en la nueva cabeza. (Si la cola se vacía, entonces el lema se mantiene vacío). Por la hipótesis inductiva,  $d(h_{i-1}) \leq d(h_i)$ . Pero entonces tenemos  $d(h_{i-1}) \leq d(h_i) \leq d(h_i)$ , y las desigualdades restantes no se ven afectadas. Así, el lema sigue con la cabeza. 2 como

Para entender lo que sucede al poner en cola un vértice, necesitamos examinar el código más de cerca. Cuando ponemos en cola un vértice en la línea 17 de BFS, se convierte en  $v$ . En ese momento, ya hemos eliminado el vértice  $u$ , cuya lista de adyacencia se está escaneando actualmente, de la cola  $Q$ , y por la hipótesis inductiva, la nueva cabeza tiene  $d(v) \leq d(u)$ . Así,  $d(v) \leq d(u) \leq d(u)$ . De la hipótesis inductiva, también tenemos  $d(v) \leq d(u)$ , por lo que  $d(v) \leq d(u) \leq d(v)$ , y las desigualdades restantes no se ven afectadas. Por lo tanto, el lema sigue cuando está en cola. ■

El siguiente corolario muestra que los valores de  $d$  en el momento en que los vértices son en cola aumentan monótonamente con el tiempo.

**Corolario 22.4**

Supongamos que los vértices  $i$  y  $j$  se ponen en cola durante la ejecución de BFS, y  $i < j$ . Entonces  $i$  se pone en cola antes que  $j$ .

Prueba Inmediata del Lema 22.3 y la propiedad de que cada vértice recibe un valor  $d$  finito como máximo una vez durante el curso de BFS. ■

Ahora podemos probar que la búsqueda primero en amplitud encuentra correctamente la ruta más corta

**Teorema 22.5 (Corrección de la búsqueda primero en amplitud)**

Sea  $G = (V, E)$  sea un grafo dirigido o no dirigido, y suponga que BFS se ejecuta en  $G$  desde un vértice fuente dado  $s \in V$ . Luego, durante su ejecución, BFS descubre cada vértice  $v \in V$  que es accesible desde la fuente  $s$ , y al terminar,  $d(v) = s$  para todos los  $v \in V$ . Además, para cualquier vértice  $v \neq s$  que sea alcanzable

de s, uno de los caminos más cortos de s a es un camino más corto de s a : seguido por el borde ..; /.

Prueba Suponga, con fines de contradicción, que algún vértice recibe un valor publicitario que no es igual a su distancia de ruta más corta. Sea el vértice con mínimo  $i.s; /$  que recibe un valor d tan incorrecto; claramente  $\neq s$ . Por el Lema 22.2,  $:d i.s; /$ , y así tenemos que  $:d > i.s; /$ . El vértice debe ser accesible desde s, porque si no lo es, entonces  $i.s; / D 1 :d$ . Sea u el vértice im tal que  $i.s; / D i.s; u/ C$  en un camino más corto de s a Porque  $i.s; u/ < \dots$ , 1. precediendo mediatamente  $i.s; /$ , y por como elegimos tenemos  $u:d D i.s; tu/ \dots$ ,  
Juntando estas propiedades, tenemos

$$:d > i.s; / D i.s; u/ C 1 D u:d C 1 \quad (22.1)$$

Ahora considere el momento en que BFS elige eliminar el vértice u de Q en la línea 11. En este momento, el vértice es blanco, gris o negro. Mostraremos que en cada uno de estos casos derivamos una contradicción a la desigualdad (22.1). Si es blanca, entonces la línea 15 establece  $:d D u:d C 1$ , contradiciendo la desigualdad (22.1). Si es negro, entonces ya fue eliminado de la cola y, por el Corolario 22.4, tenemos  $:d u:d$ , contradiciendo nuevamente la desigualdad (22.1). Si es gris, entonces se pintó de gris al quitar la cola de algún vértice w, que se eliminó de Q antes que u y para el cual  $:d D w:d C 1$ . Sin embargo, por el Corolario 22.4,  $w:d u:d$ , y así tenemos  $:d D w:d C 1 u:d C 1$ , contradiciendo una vez más la desigualdad (22.1).

Así concluimos que  $:d D i.s; /$  para todos los  $2 V$ . Todos los vértices alcanzables desde s deben ser descubiertos, porque de lo contrario tendrían  $1 D :d > i.s; /$ . Para concluir la prueba del teorema, observe que si  $D u$ , entonces  $:d D u:d C 1$ . Por lo tanto, podemos obtener un camino más corto de s a tomando un camino más corto de s a : y luego recorriendo el borde ..; /.

■

#### Árboles primero en anchura

El procedimiento BFS construye un árbol primero en anchura a medida que busca en el gráfico, como ilustra la figura 22.3. El árbol corresponde a los atributos. Más formalmente, para un grafo  $GD .V; E/$  con fuente s, definimos el subgrafo predecesor de G como  $GD .V; E/$ ,

donde  $VD f 2 VW: \neq NILg [ fsg$

y

$ED f.; / W 2 V fsgg :$

El subgrafo predecesor G es un árbol primero en anchura si V consta de los vértices alcanzables desde s y, para todos los  $2 V$ , el subgrafo G contiene un único simple

El camino de  $s$  a  $t$  también es el camino más corto de  $s$  a  $t$  en  $G$ . Un árbol primero en anchura es de hecho un árbol, ya que es conexo y  $|E| \geq |V| - 1$  (ver Teorema B.2). Llamamos a las aristas en  $E$  aristas de árbol.

El siguiente lema muestra que el subgrafo predecesor producido por el procedimiento BFS es un árbol primero en anchura.

#### Lema 22.6

Cuando se aplica a un grafo dirigido o no dirigido  $G = (V, E)$ , el procedimiento BFS construye de modo que el subgrafo predecesor  $G' = (V', E')$  es un árbol primero en anchura.

La línea de prueba 16 de BFS establece  $D[u] \neq \emptyset$  si y solo si  $u \in V'$  y  $\text{dist}(s, u) < \infty$ —es decir, si  $s$  es alcanzable desde  $s$ —y por lo tanto  $V'$  consta de los vértices en  $V$  accesibles desde  $s$ . Dado que  $G$  forma un árbol, según el teorema B.2, contiene un único camino simple desde  $s$  hasta cada vértice en  $V'$ . Al aplicar el teorema 22.5 de manera induktiva, concluimos que cada uno de esos caminos es el camino más corto en  $G$ . ■

El siguiente procedimiento imprime los vértices en la ruta más corta desde  $s$  hasta  $t$ , suponiendo que BFS ya calculó un árbol primero en anchura:

```

IMPRIMIR-RUTA.G; s; /
1 si == s
2     imprimir
s 3 otra cosa : == NINGUNO
4 imprime "ninguna ruta desde" s "a" "existe" 5 else
PRINT-PATH.G; s; /: 6 imprimir

```

Este procedimiento se ejecuta en tiempo lineal en el número de vértices del camino impreso, ya que cada llamada recursiva es para un camino un vértice más corto.

#### Ejercicios

##### 22.2-1

Muestre los valores de  $d_{st}$  que resultan de ejecutar la búsqueda primero en anchura en la gráfica dirigida de la figura 22.2(a), usando el vértice 3 como fuente.

##### 22.2-2

Muestre los valores de  $d_{st}$  que resultan de ejecutar la búsqueda primero en anchura en la gráfica no dirigida de la figura 22.3, usando el vértice  $u$  como fuente.

## 22.2-3

Demuestre que es suficiente usar un solo bit para almacenar cada color de vértice argumentando que el procedimiento BFS produciría el mismo resultado si se eliminaran las líneas 5 y 14.

## 22.2-4

¿Cuál es el tiempo de ejecución de BFS si representamos su gráfico de entrada mediante una matriz de adyacencia y modificamos el algoritmo para manejar esta forma de entrada?

## 22.2-5

Argumente que en una búsqueda primero en anchura, el valor  $u:d$  asignado a un vértice  $u$  es independiente del orden en que aparecen los vértices en cada lista de adyacencia. Utilizando la Figura 22.3 como ejemplo, muestre que el árbol primero en anchura calculado por BFS puede depender del orden dentro de las listas de adyacencia.

## 22.2-6

Dé un ejemplo de un grafo dirigido  $G = \langle V; E \rangle$ , un vértice fuente  $s \in V$  conjunto de aristas, y un árbol  $T = \langle V; E \rangle$  tal que para cada vértice  $v \in V$  en el gráfico  $G = \langle V; E \rangle$  de  $s$  a  $v$ , el único camino simple es el camino más corto en  $G$ , sin embargo, el conjunto de aristas  $E$  no se puede producir ejecutando BFS en  $G$ , sin importar cómo se ordenen los vértices en cada lista de adyacencia.

## 22.2-7

Hay dos tipos de luchadores profesionales: "cara de bebé" ("buenos") y "heels" ("malos"). Entre cualquier pareja de luchadores profesionales puede haber o no rivalidad. Supongamos que tenemos  $n$  luchadores profesionales y tenemos una lista de  $r$  parejas de luchadores para los que existen rivalidades. Proporcione un algoritmo  $O(nr)$ -tiempo que determine si es posible designar a algunos de los luchadores como babyfaces y al resto como heels, de modo que cada rivalidad sea entre un babyface y un heel. Si es posible realizar tal designación, su algoritmo debería producirla.

## 22.2-8 ?

El diámetro de un árbol  $T = \langle V; E \rangle$  se define como  $\max_{u,v \in V} d(u, v)$ , es decir, la mayor de todas las distancias de camino más corto en el árbol. Proporcione un algoritmo eficiente para calcular el diámetro de un árbol y analice el tiempo de ejecución de su algoritmo.

## 22.2-9

Sea  $G = \langle V; E \rangle$  sea un grafo conexo no dirigido. Proporcione un algoritmo  $O(|V| |E|)$ -tiempo para calcular un camino en  $G$  que atraviese cada borde en  $E$  exactamente una vez en cada dirección. Describe cómo puedes encontrar la salida de un laberinto si te dan una gran cantidad de monedas de un centavo.

---

## 22.3 Búsqueda en profundidad

La estrategia seguida por la búsqueda primero en profundidad es, como su nombre lo indica, buscar "más profundamente" en el gráfico siempre que sea posible. La búsqueda primero en profundidad explora los bordes del vértice descubierto más recientemente que todavía tiene bordes sin explorar que lo abandonan. Una vez que se han explorado todas las aristas, la búsqueda "retrocede" para explorar las aristas dejando el vértice desde el que se descubrió. Este proceso continúa hasta que hayamos descubierto todos los vértices a los que se puede acceder desde el vértice fuente original.

Si quedan vértices sin descubrir, la búsqueda primero en profundidad selecciona uno de ellos como una nueva fuente y repite la búsqueda desde esa fuente. El algoritmo repite todo este proceso hasta que ha descubierto todos los vértices.<sup>3</sup>

Como en la búsqueda primero en amplitud, cada vez que la búsqueda primero en profundidad descubre un vértice durante un escaneo de la lista de adyacencia de un vértice u ya descubierto, registra este evento asignando el atributo predecesor de : a u. A diferencia de la búsqueda primero en amplitud, cuyo subgráfico predecesor forma un árbol, el subgráfico predecesor producido por una búsqueda primero en profundidad puede estar compuesto por varios árboles, porque la búsqueda puede repetirse desde múltiples fuentes. Por lo tanto, definimos el subgrafo predecesor de una búsqueda primero en profundidad ligeramente diferente al de una búsqueda primero en amplitud: dejamos  $GD.V; E/$ , donde

$ED.f:: / W 2 V y : \bowtie NILg :$

El subgrafo predecesor de una búsqueda primero en profundidad forma un bosque primero en profundidad que comprende varios árboles primero en profundidad. Los bordes en  $E$  son bordes de árbol.

Al igual que en la búsqueda primero en amplitud, la búsqueda primero en profundidad colorea los vértices durante la búsqueda para indicar su estado. Cada vértice es inicialmente blanco, se atenúa cuando se descubre en la búsqueda y se oscurece cuando se termina, es decir, cuando se ha examinado por completo su lista de adyacencia. Esta técnica garantiza que cada vértice termine exactamente en un árbol de profundidad primero, de modo que estos árboles sean disjuntos.

Además de crear un bosque primero en profundidad, la búsqueda primero en profundidad también marca la hora de cada versión. Texas. Cada vértice tiene dos marcas de tiempo: la primera marca de tiempo : d registra cuándo se descubrió por primera vez (y se atenúa), y la segunda marca de tiempo : f registra cuándo la búsqueda termina de examinar la lista de adyacencia (y se oscurece). Estas marcas de tiempo

---

<sup>3</sup>Puede parecer arbitrario que la búsqueda primero en amplitud se limite a una sola fuente, mientras que la búsqueda primero en profundidad puede buscar en múltiples fuentes. Aunque conceptualmente, la búsqueda primero en amplitud podría provenir de múltiples fuentes y la búsqueda primero en profundidad podría limitarse a una fuente, nuestro enfoque refleja cómo se usan típicamente los resultados de estas búsquedas. La búsqueda primero en amplitud generalmente sirve para encontrar las distancias de ruta más cortas (y el subgráfico predecesor asociado) desde una fuente determinada. La búsqueda primero en profundidad es a menudo una subrutina en otro algoritmo, como veremos más adelante en este capítulo.

proporcionan información importante sobre la estructura del gráfico y, en general, son útiles para razonar sobre el comportamiento de la búsqueda en profundidad.

El siguiente procedimiento DFS registra cuándo descubre el vértice  $u$  en el atributo  $u:d$  y cuándo termina el vértice  $u$  en el atributo  $u:f$ . Estas marcas de tiempo son números enteros entre 1 y  $2|V|$ , ya que hay un evento de descubrimiento y un evento de finalización para cada uno de los vértices  $j \in V$ . Para cada vértice  $u$ ,

$$u:d < u:f : \quad (22.2)$$

El vértice  $u$  es BLANCO antes del tiempo  $u:d$ , GRIS entre el tiempo  $u:d$  y el tiempo  $u:f$ , NEGRO y a partir de entonces.

El siguiente pseudocódigo es el algoritmo básico de búsqueda en profundidad. El gráfico de entrada  $G$  puede ser dirigido o no dirigido. La variable tiempo es una variable global que usamos para la marca de tiempo.

DFS.G/ 1

```
para cada vértice  $u \in G:V$ 
     $u:\text{color} \leftarrow \text{BLANCO}$ 
     $3 \quad tu: \leftarrow \text{NIL}$ 
    4  $\text{tiempo} \leftarrow 0$ 
    5 por cada vértice  $u \in G:V$ 
    6     si  $u:\text{color} == \text{BLANCO}$ 
    7         DFS-VISIT.G;  $tu/$ 
```

DFS-VISITA.G;  $tu/$

```
1 vez  $D \text{ tiempo } C$  1 2  $u:d$  // el vértice blanco  $u$  acaba de ser descubierto
D tiempo 3  $u:\text{color}$ 
D GRIS
4 por cada  $2 \in G:\text{Adj}[u]$  // explora el borde  $.u; /$ 
5     si  $u:\text{color} == \text{BLANCO}$ 
6          $tu$ 
7         DFS-VISITA.G; /
8      $u:\text{color} \leftarrow \text{NEGRO}$  // ennegrecer  $u$ ; esta terminado
9  $\text{tiempo} \leftarrow D \text{ tiempo } C$  1
10  $u:f \leftarrow \text{tiempo}$ 
```

La Figura 22.4 ilustra el progreso de DFS en el gráfico que se muestra en la Figura 22.2.

El procedimiento DFS funciona de la siguiente manera. Las líneas 1–3 pintan todos los vértices de blanco e inicializan sus atributos en NIL. La línea 4 reinicia el contador de tiempo global. Las líneas 5 a 7 verifican cada vértice en  $V$  por turnos y, cuando se encuentra un vértice blanco, visítelo usando DFS-VISIT. Cada vez que DFS-VISIT.G;  $u/$  se llama en la línea 7, el vértice  $u$  se convierte en

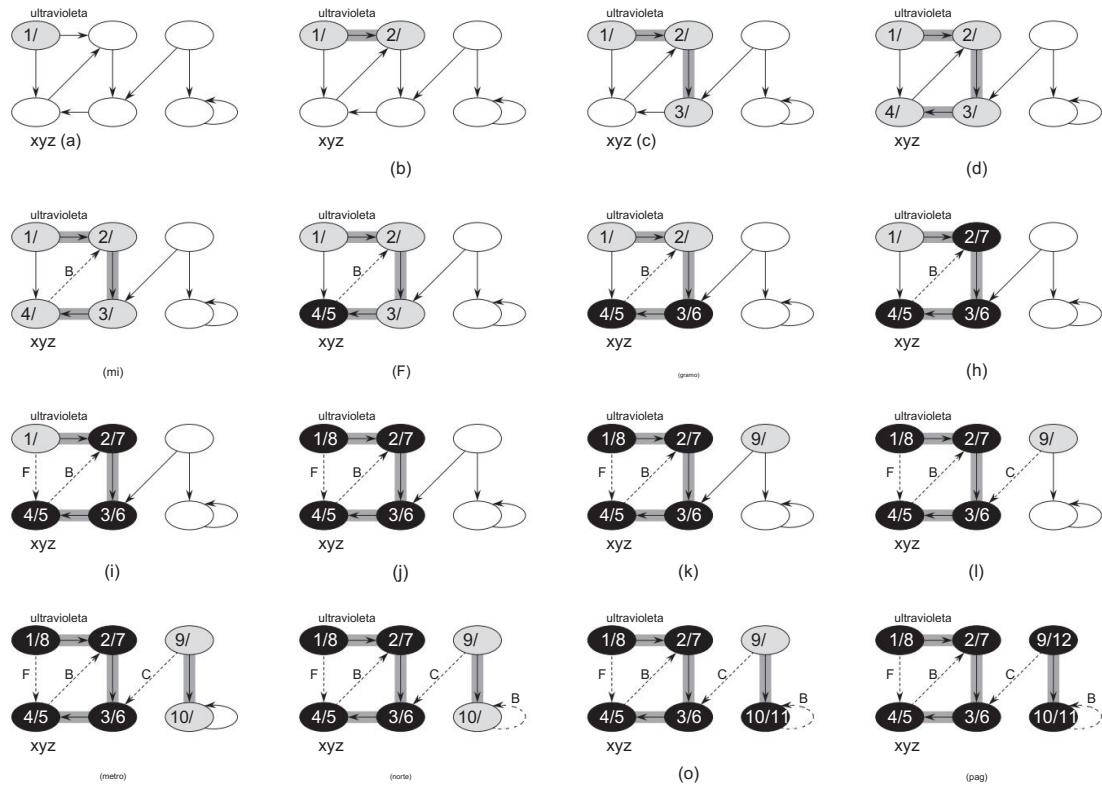


Figura 22.4 El progreso del algoritmo de búsqueda primero en profundidad DFS en un gráfico dirigido. A medida que el algoritmo explora los bordes, se muestran sombreados (si son bordes de árbol) o discontinuos (de lo contrario). Los bordes que no son árboles se etiquetan como B, C o F según sean bordes traseros, cruzados o delanteros. Las marcas de tiempo dentro de los vértices indican el tiempo de descubrimiento/tiempos de finalización.

la raíz de un nuevo árbol en el bosque de profundidad primero. Cuando DFS regresa, a cada vértice u se le ha asignado un tiempo de descubrimiento  $u:d$  y un tiempo de finalización  $u:f$ .

En cada llamada `DFS-VISIT.G; u/`, el vértice  $u$  es inicialmente blanco. La línea 1 incrementa el tiempo de la variable global  $\text{time}$ , la línea 2 registra el nuevo valor de tiempo como el tiempo de descubrimiento  $u:d$ , y la línea 3 pinta  $u$  gris. Las líneas 4 a 7 examinan cada vértice adyacente a  $u$  y visitan recursivamente si es blanco. Como cada vértice  $v$  se considera en la línea 4, decimos que arista  $u:v$  es explorado por la búsqueda primero en profundidad. Finalmente, después de explorar cada borde que sale de  $u$ , las líneas 8–10 pintan  $u$  de negro, incrementan el tiempo y registran el tiempo de finalización en  $u:f$ .

Tenga en cuenta que los resultados de la búsqueda primero en profundidad pueden depender del orden en que la línea 5 de DFS examina los vértices y del orden en que la línea 4 de DFS VISIT visita los vecinos de un vértice. Estos diferentes órdenes de visitas tienden a no

causar problemas en la práctica, ya que generalmente podemos usar cualquier resultado de búsqueda en profundidad de manera efectiva, con resultados esencialmente equivalentes.

¿Cuál es el tiempo de ejecución de DFS? Los bucles de las líneas 1 a 3 y de las líneas 5 a 7 de DFS toman un tiempo  $,V /$ , exclusivo del tiempo para ejecutar las llamadas a DFS-VISIT. Al igual que hicimos con la búsqueda primero en amplitud, utilizamos el análisis agregado. El procedimiento DFS-VISIT se llama exactamente una vez para cada vértice  $V$  ya que el vértice  $u$  en el que se invoca DFS-VISIT debe ser blanco y lo primero que hace DFS-VISIT es pintar el vértice  $u$  de gris. Durante una ejecución de DFS-VISIT.G; /, el ciclo en las líneas 4–7 ejecuta  $jAdj\mathcal{E}j$  veces. Desde

```
X jAdj\mathcal{E}j D ..E/ ;
2V
```

el costo total de ejecución de las líneas 4 a 7 de DFS-VISIT es  $,E/$ . Por lo tanto, el tiempo de ejecución de DFS es  $,VCE/$ .

#### Propiedades de la búsqueda primero en profundidad

La búsqueda en profundidad proporciona información valiosa sobre la estructura de un gráfico. Quizás la propiedad más básica de la búsqueda primero en profundidad es que el subgráfico predecesor  $G$  forma un bosque de árboles, ya que la estructura de los primeros árboles en profundidad refleja exactamente la estructura de las llamadas recursivas de DFS-VISIT . Es decir, tu  $D$  . . . si y solo si DFS-VISIT.G; Me llamaron durante una búsqueda en la lista de publicidad de  $u$ . Además, el vértice es un descendiente del vértice  $u$  en el bosque de profundidad primero si y solo si se descubre durante el tiempo en que  $u$  es gris.

Otra propiedad importante de la búsqueda en profundidad es que los tiempos de descubrimiento y finalización tienen una estructura de paréntesis. Si representamos el descubrimiento del vértice  $u$  con un paréntesis izquierdo “. $u$ ” y su terminación con un paréntesis derecho “ $u/$ ”, entonces la historia de descubrimientos y terminaciones forma una expresión bien formada en el sentido de que los paréntesis son propiamente anidado. Por ejemplo, la búsqueda primero en profundidad de la figura 22.5(a) corresponde al paréntesis que se muestra en la figura 22.5(b). El siguiente teorema proporciona otra forma de caracterizar la estructura de paréntesis.

#### Teorema 22.7 (teorema de paréntesis)

En cualquier búsqueda en profundidad de un gráfico (dirigido o no dirigido)  $GD .V; E/$ , para cualesquiera dos vértices  $u$  y  $f$ , se cumple exactamente una de las siguientes tres condiciones:

los intervalos  $\mathcal{E}u:d; u:f$  y  $\mathcal{E}d:f$  son completamente disjuntos, y ni  $u$  ni  $f$  son descendientes del otro en el bosque de profundidad primero, el intervalo

$\mathcal{E}u:d; u:f$  está contenido enteramente dentro del intervalo  $\mathcal{E}d:f$ ,  $yu$  es un descendiente de en un árbol primero en profundidad, o el intervalo

$\mathcal{E}d:f$  está contenido enteramente dentro del intervalo  $\mathcal{E}u:d; u:f$ , y es un descendiente de  $u$  en un árbol de profundidad primero.

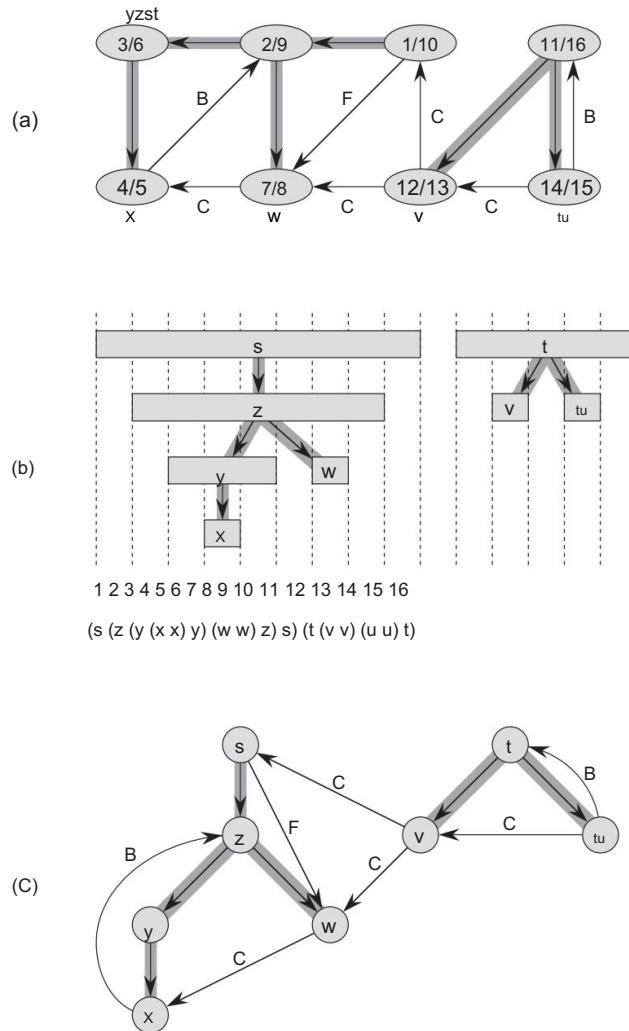


Figura 22.5 Propiedades de la búsqueda primero en profundidad. (a) El resultado de una búsqueda en profundidad de un gráfico dirigido. Los vértices tienen marca de tiempo y los tipos de borde se indican como en la Figura 22.4. (b) Los intervalos para el tiempo de descubrimiento y el tiempo de finalización de cada vértice corresponden al paréntesis que se muestra. Cada rectángulo abarca el intervalo dado por los tiempos de descubrimiento y finalización del vértice correspondiente.

Solo se muestran los bordes de los árboles. Si dos intervalos se superponen, entonces uno está anidado dentro del otro, y el vértice correspondiente al intervalo más pequeño es descendiente del vértice correspondiente al más grande. (c) El gráfico de la parte (a) redibujado con todos los árboles y los bordes delanteros descendiendo dentro de un árbol de profundidad primero y todos los bordes traseros subiendo desde un descendiente hasta un antepasado.

Prueba Comenzamos con el caso en que  $u:d < :d$ . Consideramos dos subcasos, según que  $:d < u:f$  o no. El primer subcaso ocurre cuando  $:d < u:f$ , por lo que se descubrió cuando  $u$  aún era gris, lo que implica que es descendiente de  $u$ . Además, dado que se descubrió más recientemente que  $u$ , todos sus bordes salientes se exploran y finalizan antes de que la búsqueda regrese y finalice  $u$ . En este caso, por tanto, el intervalo  $\langle :d; :f \rangle$  está completamente contenido dentro del intervalo  $\langle :u:d; :u:f \rangle$ . En el otro subcaso,  $u:f < :d$ , y por desigualdad (22.2),  $u:d < u:f < :d < :f$ ; así los intervalos  $\langle :u:d; :u:f \rangle$  y  $\langle :d; :f \rangle$  son disjuntos.

Debido a que los intervalos son disjuntos, no se descubrió ningún vértice mientras que el otro era gris, por lo que ninguno de los vértices es descendiente del otro.

El caso en el que  $:d < u:d$  es similar, con los roles de  $u$  y revertidos en el argumento anterior. ■

#### Corolario 22.8 (Anidamiento de intervalos de descendientes)

El vértice  $v$  es un descendiente propio del vértice  $u$  en el bosque primero en profundidad para un gráfico  $G$  (dirigido o no dirigido) si y sólo si  $u:d < :d < :f < u:f$ .

Prueba inmediata del teorema 22.7. ■

El siguiente teorema da otra caracterización importante de cuando un vértice es descendiente de otro en el bosque de profundidad primero.

#### Teorema 22.9 (Teorema del camino blanco)

En un bosque primero en profundidad de un grafo (dirigido o no dirigido)  $G$  . $V; E$ , el vértice  $w$  es un descendiente del vértice  $u$  si y sólo si en el momento  $u:d$  en que la búsqueda descubre  $u$ , hay un camino desde  $u$  hasta  $w$  que consta enteramente de vértices blancos.

Prueba ): Si  $D_u$ , entonces el camino desde  $u$  hasta contiene solo el vértice  $u$ , que sigue siendo blanco cuando establecemos el valor de  $u:d$ . Ahora, supongamos que es un descendiente propio de  $u$  en el bosque primero en profundidad. Por el Corolario 22.8,  $u:d < :d$ , y así es blanco en el tiempo  $u:d$ . Dado que puede ser cualquier descendiente de  $u$ , todos los vértices en el único camino simple desde  $u$  hasta el primer bosque en profundidad son blancos en el momento  $u:d$ .

(: Suponga que hay un camino de vértices blancos desde  $u$  hasta en el tiempo  $u:d$ , pero no se convierte en un descendiente de  $u$  en el árbol de profundidad primero. Sin pérdida de generalidad, suponga que todos los vértices que no sean a lo largo del camino pasa a ser descendiente de  $u$ . (De lo contrario, sea el vértice más cercano a  $u$  a lo largo del camino que no se convierte en un descendiente de  $u$ ). Sea  $w$  el predecesor de en el camino, de modo que  $w$  sea un descendiente de  $u$  ( $w$  y  $u$  pueden de hecho ser el mismo vértice). Por el Corolario 22.8,  $w:f < u:f$ . Debido a que debe descubrirse después de descubrir  $u$ , pero antes de que  $w$  termine, tenemos  $u:d < :d < w:f < u:f$ . El teorema 22.7 implica entonces que el intervalo  $\langle :d; :f \rangle$

está contenido enteramente dentro del intervalo  $\{u:d; u: f\}$ . Por el Corolario 22.8, debe después de todo ser descendiente de  $u$ .

#### Clasificación de bordes

Otra propiedad interesante de la búsqueda primero en profundidad es que la búsqueda se puede utilizar para clasificar los bordes del gráfico de entrada  $G = \langle V, M \rangle$ . El tipo de cada borde puede proporcionar información importante sobre un gráfico. Por ejemplo, en la siguiente sección, veremos que un grafo dirigido es acíclico si y solo si una búsqueda en profundidad no produce bordes "atrás" (Lema 22.11).

Podemos definir cuatro tipos de borde en términos del bosque de profundidad primero  $G$  producido por una búsqueda en profundidad primero en  $G$ :

1. Los bordes de los árboles son bordes en el bosque de profundidad primero  $G$ . Edge  $.u; /$  es un borde de árbol si fue descubierto por primera vez al explorar edge  $.u; /$ .
2. Los bordes posteriores son aquellos bordes  $.u; /$  conectando un vértice  $u$  a un antepasado en un árbol de profundidad primero. Consideramos que los bucles automáticos, que pueden ocurrir en gráficos dirigidos, son bordes posteriores.
3. Los bordes delanteros son aquellos bordes que no son de árbol  $.u; /$  conectando un vértice  $u$  con un descendiente en un árbol primero en profundidad.
4. Los bordes cruzados son todos los demás bordes. Pueden ir entre vértices en el mismo árbol primero en profundidad, siempre que un vértice no sea un ancestro del otro, o pueden ir entre vértices en diferentes árboles primero en profundidad.

En las Figuras 22.4 y 22.5, las etiquetas de los bordes indican los tipos de bordes. La figura 22.5(c) también muestra cómo volver a dibujar el gráfico de la figura 22.5(a) de modo que todos los bordes del árbol y delanteros se dirijan hacia abajo en un árbol de profundidad primero y todos los bordes traseros suban. Podemos redibujar cualquier gráfico de esta manera.

El algoritmo DFS tiene suficiente información para clasificar algunos bordes a medida que los encuentra. La idea clave es que cuando exploramos por primera vez un borde  $.u; /$ , el color del vértice nos dice algo sobre el borde:

1. BLANCO indica el borde de un árbol, 2.

GRIS indica un borde trasero y 3. NEGRO indica un borde delantero o transversal.

El primer caso es inmediato a partir de la especificación del algoritmo. Para el segundo caso, observe que los vértices grises siempre forman una cadena lineal de descendientes correspondientes a la pila de invocaciones DFS-VISIT activas; el número de vértices grises es uno más que la profundidad en el bosque de profundidad primero del vértice descubierto más recientemente. La exploración siempre procede del vértice gris más profundo, por lo que

una arista que llega a otro vértice gris ha llegado a un ancestro. El tercer caso maneja la posibilidad restante; El ejercicio 22.3-5 le pide que demuestre que tal arista  $.u; / es$  un borde delantero si  $u:d < :d$  y un borde cruzado si  $u:d > :d$ .

Un grafo no dirigido puede implicar cierta ambigüedad en la clasificación de las aristas, ya que  $.u; / y ; u /$  son realmente el mismo borde. En tal caso, clasificamos el borde como el primer tipo en la lista de clasificación que se aplica. De manera equivalente (vea el ejercicio 22.3-6), clasificamos la arista según cualquiera de  $.u; / o ; tu/ el$   
Buscar encuentros primero.

Ahora mostramos que los bordes delanteros y cruzados nunca ocurren en una búsqueda primero en profundidad de un grafo no dirigido.

#### Teorema 22.10 En

una búsqueda en profundidad de un grafo  $G$  no dirigido, cada arista de  $G$  es una arista de árbol o una arista posterior.

Prueba Sea  $.u; /$  sea una arista arbitraria de  $G$ , y suponga sin pérdida de generalidad que  $u:d < :d$ . Entonces la búsqueda debe descubrir y terminar antes de que termine  $u$  (mientras que  $u$  es gris), ya que está en la lista de adyacencia de  $u$ . Si la primera vez que la búsqueda explora edge  $.u; /$ , está en la dirección de  $u$  hasta entonces no se ha descubierto (blanco) hasta ese momento, porque de lo contrario la búsqueda ya habría explorado este borde en la dirección de  $u$ . Así,  $.u; /$  se convierte en el borde de un árbol. Si la búsqueda explora  $.u; /$  primero en la dirección de  $u$ , luego  $.u; /$  es un borde posterior, ya que  $u$  todavía es gris en el momento en que se explora el borde por primera vez. ■

Veremos varias aplicaciones de estos teoremas en las siguientes secciones.

### Ejercicios

#### 22.3-1

Haga un gráfico de 3 por 3 con etiquetas de fila y columna BLANCO, GRIS y NEGRO. En cada celda  $i; j/$ , indican si, en cualquier punto durante una búsqueda en profundidad de un gráfico dirigido, puede haber una arista desde un vértice de color  $i$  hasta un vértice de color  $j$ .

Para cada posible borde, indique qué tipos de borde pueden ser. Cree un segundo gráfico de este tipo para la búsqueda en profundidad de un gráfico no dirigido.

#### 22.3-2

Muestre cómo funciona la búsqueda primero en profundidad en el gráfico de la figura 22.6. Suponga que el bucle for de las líneas 5 a 7 del procedimiento DFS considera los vértices en orden alfabético y suponga que cada lista de adyacencia está ordenada alfabéticamente. Muestre los tiempos de descubrimiento y finalización para cada vértice, y muestre la clasificación de cada borde.

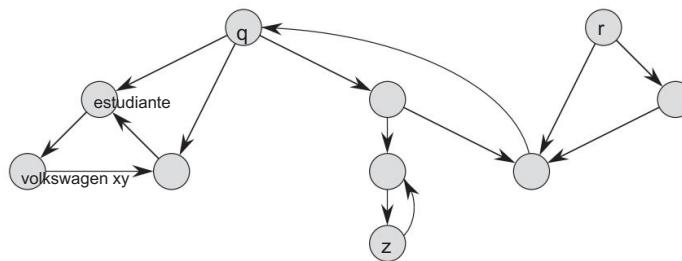


Figura 22.6 Gráfica dirigida para usar en los ejercicios 22.3-2 y 22.5-2.

22.3-3

Muestre la estructura de paréntesis de la búsqueda en profundidad de la figura 22.4.

22.3-4

Demuestre que es suficiente usar un solo bit para almacenar cada color de vértice argumentando que el procedimiento DFS produciría el mismo resultado si se eliminara la línea 3 de DFS-VISIT.

22.3-5

Muestre esa arista  $.u; / es$ 

- un borde de árbol o borde delantero si y solo si  $u:d < :d < :f < u:f$ ,
- un borde posterior si y solo si  $:d < u:d < u:f$
- un borde cruzado si y solo si  $:d < :f < u:d < u:f$ .

22.3-6

Muestre que en un grafo no dirigido, clasificando una arista  $.u; /$  como borde de árbol o borde posterior según sea  $.u; / o ; u/$  se encuentra primero durante la búsqueda en profundidad es equivalente a clasificarlo según el orden de los cuatro tipos en el esquema de clasificación.

22.3-7

Vuelva a escribir el procedimiento DFS, utilizando una pila para eliminar la recursividad.

22.3-8

Dé un contraejemplo a la conjetura de que si un grafo dirigido  $G$  contiene un camino de  $u$  a  $v$  y si  $u:d < :d$  en una búsqueda de  $G$  primero en profundidad, entonces  $v$  es un descendiente de  $u$  en el bosque primero en profundidad producido.

## 22.3-9

Dé un contraejemplo a la conjetura de que si un grafo dirigido  $G$  contiene un camino de  $u$  a  $v$ , entonces cualquier búsqueda en profundidad debe dar como resultado  $:d u:v$ .

## 22.3-10

Modifique el pseudocódigo para la búsqueda primero en profundidad para que imprima cada borde en el gráfico dirigido  $G$ , junto con su tipo. Muestre qué modificaciones, si las hay, necesita hacer si  $G$  no está dirigida.

## 22.3-11

Explique cómo un vértice  $u$  de un grafo dirigido puede terminar en un árbol primero en profundidad que contiene solo  $u$ , aunque  $u$  tiene aristas entrantes y salientes en  $G$ .

## 22.3-12

Demuestre que podemos usar una búsqueda en profundidad de un grafo no dirigido  $G$  para identificar las componentes conexas de  $G$ , y que el bosque en profundidad contiene tantos árboles como componentes conexas tiene  $G$ . Más precisamente, muestre cómo modificar la búsqueda primero en profundidad para que asigne a cada vértice una etiqueta de número entero  $:cc$  entre 1 y  $k$ , donde  $k$  es el número de componentes conectadas de  $G$ , tal que  $u:cc \neq v:cc$  si y solo si  $u$  y  $v$  están en la misma componente conexa.

## 22.3-13 ?

Un grafo dirigido  $G$  es conexo si  $\dots$  implica que  $G$  contiene como máximo un camino simple desde  $u$  hasta para todos los vértices  $u; 2$  voltios Proporcione un algoritmo eficiente para determinar si un grafo dirigido es conexo o no.

## 22.4 Clasificación topológica

Esta sección muestra cómo podemos usar la búsqueda primero en profundidad para realizar una especie topológica de un gráfico acíclico dirigido, o un "dag", como a veces se le llama. Una especie topológica de un dag  $G$  es una ordenación lineal de todos sus vértices tal que si  $G$  contiene una arista  $u; v$ , entonces  $u$  aparece antes en el ordenamiento. (Si el gráfico contiene un ciclo, entonces no es posible un ordenamiento lineal). Podemos ver un tipo topológico de un gráfico como un ordenamiento de sus vértices a lo largo de una línea horizontal de modo que todos los bordes dirigidos van de izquierda a derecha. La clasificación topológica es, por lo tanto, diferente del tipo habitual de "clasificación" estudiado en la Parte II.

Muchas aplicaciones utilizan gráficos acíclicos dirigidos para indicar precedencias entre eventos. La figura 22.7 da un ejemplo que surge cuando el profesor Bumstead se viste por la mañana. El profesor debe ponerse ciertas prendas antes que otras (por ejemplo, calcetines antes que zapatos). Se pueden poner otros artículos en cualquier orden (p. ej., calcetines y

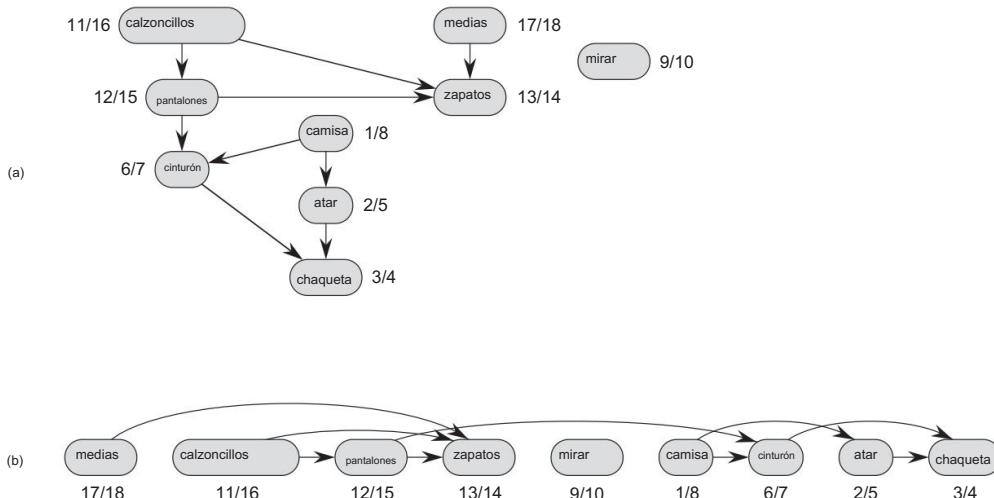


Figura 22.7 (a) El profesor Burnstead clasifica topológicamente su ropa cuando se viste. Cada arista dirigida  $.u; /$  significa que la prenda debe ponerse antes que la prenda  $.u$ . Los tiempos de descubrimiento y finalización de una búsqueda en profundidad se muestran junto a cada vértice. (b) El mismo gráfico que se muestra ordenado topológicamente, con sus vértices dispuestos de izquierda a derecha en orden decreciente de tiempo de finalización.

Todos los bordes dirigidos van de izquierda a derecha.

pantalones). Un borde dirigido  $.u; /$  en el dag de la figura 22.7(a) indica que la prenda  $u$  debe ponerse antes que la prenda  $u$ .

Una especie topológica de este dag, por tanto, da una orden

para vestirse. La figura 22.7(b) muestra el dag clasificado topológicamente como una ordenación de vértices a lo largo de una línea horizontal de modo que todos los bordes dirigidos van de izquierda a derecha.

El siguiente algoritmo simple ordena topológicamente un dag:

**TOPOLOGICAL-SORT.G/ 1** llama

a DFS.G/ para calcular los tiempos de terminación :f para cada vértice 2 cuando cada vértice está terminado, insértelo al frente de una lista enlazada 3 devuelve la lista enlazada de vértices

La Figura 22.7(b) muestra cómo los vértices clasificados topológicamente aparecen en orden inverso a sus tiempos de finalización.

Podemos realizar una ordenación topológica en el tiempo „VCE“, ya que la búsqueda en profundidad toma „VCE“ tiempo y toma O.1/ tiempo para insertar cada uno de los vértices JV j al frente de la lista enlazada.

Probamos la corrección de este algoritmo usando el siguiente lema clave char  
Caracterización de grafos acíclicos dirigidos.

**Lema 22.11 Un**

grafo dirigido  $G$  es acíclico si y sólo si una búsqueda de  $G$  en profundidad no produce aristas posteriores.

Prueba ): suponga que una búsqueda en profundidad produce un borde posterior  $.u; /$ . Entonces el vértice es un ancestro del vértice  $u$  en el bosque de profundidad primero. Por lo tanto,  $G$  contiene un camino desde  $a u$ , y el borde posterior  $.u; /$  completa un ciclo.

(: Suponga que  $G$  contiene un ciclo  $c$ . Demostramos que una búsqueda de  $G$  primero en profundidad produce un borde posterior. Sea el primer vértice que se descubre en  $c$ , y sea  $.u; /$  el borde anterior en  $c$ . En tiempo  $:d$ , los vértices de  $c$  forman un camino de vértices blancos desde a  $u$ . Por el teorema del camino blanco, el vértice  $u$  se convierte en un descendiente de en el bosque primero en profundidad. Por lo tanto,  $.u; /$  es un borde posterior. ■

**El teorema 22.12**

CLASIFICACIÓN TOPOLOGICA produce una clasificación topológica del grafo acíclico dirigido proporcionado como entrada.

Prueba Suponga que DFS se ejecuta en un dag  $GD$  .V dado; E/ para determinar los tiempos de terminación de sus vértices. Basta mostrar que para cualquier par de vértices distintos  $u; 2 V$  entonces  $:f < u:f$  .~~Consideremos que la arista  $u; /$  ha sido explorada por DFS. G/~~ Cuando se explora este borde, no puede ser gris, ya que entonces sería un ancestro de  $u$  y  $.u; /$  sería un borde posterior, contradiciendo el Lema 22.11. Por lo tanto, debe ser blanco o negro. Si es blanco, se convierte en descendiente de  $u$ , y así  $:f < u:f$  .

Si es negro, ya se ha terminado, por lo que ya se ha configurado  $:f$  . Debido a que todavía estamos explorando desde  $u$ , todavía tenemos que asignar una marca de tiempo a  $u:f$  , y una vez que lo hagamos, también tendremos  $:f < u:f$  . Así, para cualquier arista  $.u; /$  en el dag, tenemos  $:f < u:f$  , probando el teorema. ■

**Ejercicios****22.4-1**

Muestre el orden de los vértices producido por TOPOLOGICAL-SORT cuando se ejecuta en el dag de la figura 22.8, bajo el supuesto del ejercicio 22.3-2.

**22.4-2**

Proporcione un algoritmo de tiempo lineal que tome como entrada un gráfico acíclico dirigido  $GD$  .V; E/ y dos vértices  $s$  y  $t$ , y devuelve el número de caminos simples de  $s$  a  $t$  en  $G$ . Por ejemplo, el gráfico acíclico dirigido de la figura 22.8 contiene exactamente cuatro caminos simples del vértice  $p$  al vértice: po, pory, posry, y psry.

(Su algoritmo solo necesita contar las rutas simples, no enumerarlas).

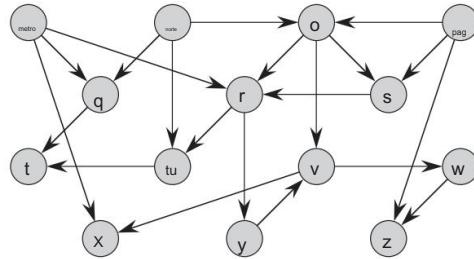


Figura 22.8 Un dag para clasificación topológica.

## 22.4-3

Dé un algoritmo que determine si un grafo no dirigido dado  $G = \langle V, E \rangle$  contiene un ciclo. Su algoritmo debe ejecutarse en  $O(V)$  tiempo, independientemente de  $|E|$ .

## 22.4-4

Demostrar o refutar: si un grafo dirigido  $G$  contiene ciclos, entonces TOPOLOGICAL SORT. $G$ / produce una ordenación de vértices que minimiza el número de aristas “incorrectas” que son inconsistentes con la ordenación producida.

## 22.4-5

Otra forma de realizar la ordenación topológica en un grafo acíclico dirigido  $G = \langle V, E \rangle$  es encontrar repetidamente un vértice de grado 0, generarlo y eliminarlo y todos sus bordes salientes del gráfico. Explique cómo implementar esta idea para que se ejecute en el tiempo  $O(V + CE)$ . ¿Qué le sucede a este algoritmo si  $G$  tiene ciclos?

## 22.5 Componentes fuertemente conectados

Ahora consideraremos una aplicación clásica de la búsqueda primero en profundidad: descomponer un gráfico dirigido en sus componentes fuertemente conectados. Esta sección muestra cómo hacerlo mediante dos búsquedas en profundidad. Muchos algoritmos que funcionan con grafos dirigidos comienzan con tal descomposición. Después de descomponer el gráfico en componentes fuertemente conectados, dichos algoritmos se ejecutan por separado en cada uno y luego combinan las soluciones de acuerdo con la estructura de conexiones entre los componentes.

Recuerde del Apéndice B que un componente fuertemente conexo de un gráfico dirigido  $G = \langle V, E \rangle$  es un conjunto máximo de vértices  $C$  tal que para cada par de vértices  $u$  y  $v$  en  $C$ , tenemos  $u \rightarrow v$  y  $v \rightarrow u$ ; es decir, los vértices  $u$  y  $v$  son alcanzables entre sí. La figura 22.9 muestra un ejemplo.

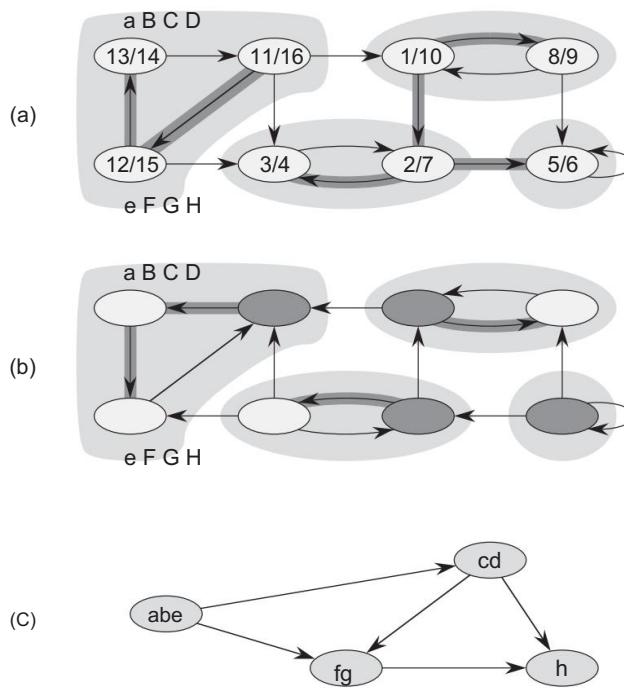


Figura 22.9 (a) Un gráfico dirigido G. Cada región sombreada es un componente fuertemente conexo de G. Cada vértice está etiquetado con sus tiempos de descubrimiento y finalización en una búsqueda en profundidad, y los bordes del árbol están sombreados. (b) El gráfico GT, la transpuesta de G, con el bosque primero en profundidad calculado en la línea 3 de COMPONENTES FUERTEMENTE CONECTADOS mostrados y los bordes de los árboles sombreados. Cada componente fuertemente conectado corresponde a un árbol primero en profundidad. Los vértices b, c, g y h, que están muy sombreados, son las raíces de los árboles primero en profundidad producidos por la búsqueda primero en profundidad de GT. (c) El gráfico de componente acíclico GSCC obtenido al contraer todos los bordes dentro de cada componente fuertemente conectado de G de modo que solo quede un vértice en cada componente.

Nuestro algoritmo para encontrar componentes fuertemente conectados de un grafo GD .V; E/ usa la transpuesta de G, que definimos en el ejercicio 22.1-3 como la gráfica GT D .V; ET/, donde ET D fu; / W ; u/ 2 Ej. Es decir, ET consta de las aristas de G con sus direcciones invertidas. Dada una representación de lista de adyacencia de G, el tiempo para crear GT es OV CE/. Es interesante observar que G y GT tienen exactamente las mismas componentes fuertemente conectadas: u y son alcanzables entre sí en G si y solo si son alcanzables entre sí en GT.

La figura 22.9(b) muestra la transposición del gráfico de la figura 22.9(a), con las componentes fuertemente conectadas sombreadas.

El siguiente algoritmo de tiempo lineal (es decir,  $\sim V \cdot CE/\text{tiempo}$ ) calcula los componentes fuertemente conectados de un grafo dirigido  $G = (V, E)$  utilizando dos búsquedas en profundidad, una en  $G$  y otra en  $GT$ .

#### COMPONENTES FUERTEMENTE CONECTADOS.G/1

Llame a  $DFS.G/$  para calcular los tiempos de finalización  $u:f$  para cada vértice  $u$

calcule  $GT$  3 llame

a  $DFS.GT/$ , pero en el ciclo principal de DFS, considere los vértices en orden

decreciente  $u:f$  (como se calcula en la línea 1) 4 genera los

vértices de cada árbol en el bosque de profundidad primero formado en la línea 3 como un componente separado fuertemente conectado

La idea detrás de este algoritmo proviene de una propiedad clave del gráfico de componentes GS<sub>CC</sub>  $D = (V, E)$ , que definimos de la siguiente manera. Suponga que  $G$  tiene componentes  $C_1, C_2, \dots, C_k$ . El conjunto de vértices  $V$  es  $f_1, f_2, \dots, f_k$ , y contiene un vértice  $f_i$  para cada componente fuertemente conexa  $C_i$  de  $G$ . Hay una arista  $(f_i, f_j) \in E$  si  $G$  contiene una arista dirigida  $(x, y)$  para alguna  $x \in C_i$  y alguna  $y \in C_j$ . Visto de otra forma, al contraer todas las aristas cuyos vértices incidentes están dentro de la misma componente fuertemente conexa de  $G$ , el grafo resultante es GS<sub>CC</sub>. La figura 22.9(c) muestra la gráfica de componentes de la gráfica de la figura 22.9(a).

La propiedad clave es que el gráfico de componentes es un dag, lo que implica el siguiente lema.

#### Lema 22.13 Sean

$C$  y  $C_0$  componentes distintas fuertemente conexas en el grafo dirigido  $G = (V, E)$ , déjanos; 2  $C$ , sea  $u_0$  y suponga que  $G$  contiene un camino  $u^0 - C_0 - u_0$ .

Entonces  $G$  tampoco puede contener un camino  $u^0 - C - u^0$ .

Prueba Si  $G$  contiene un camino, entonces contiene los caminos  $u^0 - u_0 - y - u^0$ . Por lo tanto,  $u^0$  y  $u^0$  son accesibles entre sí, lo que contradice la suposición de que  $C$  y  $C_0$  son componentes distintas fuertemente conectadas. ■

Veremos que al considerar los vértices en la segunda búsqueda primero en profundidad en orden decreciente de los tiempos de finalización que se calcularon en la primera búsqueda primero en profundidad, estamos, en esencia, visitando los vértices del gráfico de componentes (cada uno de los cuales corresponde a un componente fuertemente conectado de  $G$ ) en orden clasificado topológicamente.

Debido a que el procedimiento COMPONENTES FUERTEMENTE CONECTADOS realiza dos búsquedas en profundidad, existe la posibilidad de ambigüedad cuando discutimos  $u:d$  o  $u:f$ .

En esta sección, estos valores siempre se refieren a los tiempos de descubrimiento y finalización calculados por la primera llamada de DFS, en la línea 1.

Extendemos la notación para los tiempos de descubrimiento y finalización a conjuntos de vértices. Si  $UV$  entonces definimos  $dU/D \min_{u \in U} f_u : dG$  y  $f . U / D \max_{u \in U} f_u : f_g$ .

Es decir,  $dU / f . U /$  son el tiempo de descubrimiento más temprano y el tiempo de finalización más tardío, respectivamente, de cualquier vértice en  $U$ .

El siguiente lema y su corolario dan una propiedad clave que se relaciona fuertemente con componentes conectados y tiempos de terminación en la primera búsqueda en profundidad.

#### Lema 22.14

Sean  $C$  y  $C_0$  componentes distintas fuertemente conexas en el grafo dirigido  $GD .V; MI/$ .

Supongamos que hay una arista  $.u; / 2 E$ , donde  $u \in C$  y  $2 C_0$ . Entonces  $f . C / > f . C_0 /$ .

Demostración Consideramos dos casos, dependiendo de qué componente fuertemente conexo,  $C$  o  $C_0$  Si  $dC / < f . C_0 /$ , tuvo el primer vértice descubierto durante la búsqueda en profundidad.

$< d.C_0 /$ , sea  $x$  el primer vértice descubierto en  $C$ . En el tiempo  $x:d$ , todos los vértices en  $C$  y  $C_0$  son blancos. En ese momento,  $G$  contiene un camino desde  $x$  hasta cada vértice en  $C$  que consta solo de vértices blancos. Porque  $.u; / 2 E$ , para cualquier vértice  $w \in C_0$  también hay un camino en  $G$  en el tiempo  $x:d$  de  $x$  a  $w$  que consta solo de vértices blancos:  $x \dots u \dots w$ . Por el teorema del camino blanco, todos los vértices en  $C$  y  $C_0$  se vuelven descendientes de  $x$  en el árbol de profundidad primero. Por el Corolario 22.8,  $x$  tiene el tiempo de finalización más tardío de cualquiera de sus descendientes, por lo que  $x:f D f .C / > f .C_0 /$ .

Si en cambio tenemos  $dC / > d.C_0 /$ , sea  $y$  el primer vértice descubierto en  $C_0$ .

En el momento  $y:d$ , todos los vértices en  $C_0$  son blancos y  $G$  contiene un camino desde  $y$  hasta cada vértice en  $C_0$  que consta solo de vértices blancos. Por el teorema del camino blanco, todos los vértices en  $C_0$  se vuelven descendientes de  $y$  en el árbol de profundidad primero, y por el Corolario 22.8,  $y:f D f .C_0 /$ . En el momento  $y:d$ , todos los vértices de  $C$  son blancos. Como hay una arista  $.u; /$  de  $C$  a  $C_0$  El lema 22.13 implica que no puede haber un camino de  $C_0$  a  $C$ .

Por lo tanto, ningún vértice en  $C$  es accesible desde  $y$ . En el momento  $y:f$ , por lo tanto, todos los vértices de  $C$  siguen siendo blancos. Así, para cualquier vértice  $w \in C$ , tenemos  $w:f > y:f$ , lo que implica que  $f .C / > f .C_0 /$ . ■

El siguiente corolario nos dice que cada arista en  $GT$  que va entre diferentes componentes fuertemente conectados va desde un componente con un tiempo de finalización anterior (en la primera búsqueda en profundidad) a un componente con un tiempo de finalización posterior.

#### Corolario 22.15

Sean  $C$  y  $C_0$  componentes distintas fuertemente conexas en el grafo dirigido  $GD .V; MI/$ .

Supongamos que hay una arista  $.u; / 2 ET$ , donde  $u \in C$  y  $2 C_0$ . Entonces  $f .C / < f .C_0 /$ .

Prueba Desde  $.u; / 2$  ET, tenemos  $.; u/ 2$  E. Como las componentes fuertemente conexas de G y GT son las mismas, el Lema 22.14 implica que  $f.C / < f.C0 /$ .

■

El corolario 22.15 proporciona la clave para entender por qué funciona el algoritmo de componentes fuertemente conectados. Examinemos lo que sucede cuando realizamos la segunda búsqueda en profundidad, que está en GT. Empezamos con la componente fuertemente conexa C cuyo tiempo de finalización  $f.C$  es máximo. La búsqueda comienza desde algún vértice  $x \in C$  y visita todos los vértices en C. Por el corolario 22.15, GT no contiene aristas desde C a ningún otro componente fuertemente conectado, por lo que la búsqueda desde x no visitará los vértices en ningún otro componente. Por lo tanto, el árbol con raíz en x contiene exactamente los vértices de C. Habiendo completado la visita a todos los vértices en C, la búsqueda en la línea 3 selecciona como raíz un vértice de algún otro componente fuertemente conectado  $C_0$  cuyo tiempo de terminación  $f.C_0$  es máximo sobre todos los componentes que no sean C. De nuevo, la búsqueda visitará todos los vértices en  $C_0$  pero por el Corolario 22.15, los únicos bordes en GT desde  $C_0$  a cualquier otro componente deben ser a C, que ya hemos visitado. En general, cuando la búsqueda en profundidad de GT en la línea 3 visita cualquier componente fuertemente conectado, cualquier borde que salga de ese componente debe ser hacia componentes que la búsqueda ya visitó. Cada árbol primero en profundidad, por lo tanto, será exactamente un componente fuertemente conectado. El siguiente teorema formaliza este argumento.

#### Teorema 22.16 El

procedimiento COMPONENTES FUERTEMENTE CONECTADOS calcula correctamente las componentes fuertemente conectadas del grafo dirigido G proporcionado como entrada.

Prueba Argumentamos por inducción sobre el número de árboles primero en profundidad encontrados en la búsqueda primero en profundidad de GT en la línea 3 que los vértices de cada árbol forman un componente fuertemente conectado. La hipótesis inductiva es que los primeros  $k$  árboles producidos en la línea 3 son componentes fuertemente conectados. La base para la inducción, cuando  $k = 0$ , es trivial.

En el paso inductivo, asumimos que cada uno de los primeros  $k$  primeros árboles en profundidad producidos en la línea 3 es un componente fuertemente conectado, y consideramos el árbol  $.k$  producido. Sea la raíz de este árbol el vértice  $u$ , y sea  $u$  el componente fuertemente conexo C. Debido a cómo elegimos las raíces en la búsqueda en profundidad en la línea 3,  $u:f.D / f.C / > f.C_0 /$  para cualquier componente  $C_0$  fuertemente conectado que no sea C que aún no se haya visitado. Por la hipótesis inductiva, en el momento en que la búsqueda visita u, todos los demás vértices de C son blancos. Por el teorema del camino blanco, por lo tanto, todos los demás vértices de C son descendientes de u en su árbol de profundidad primero. Además, por la hipótesis inductiva y por el Corolario 22.15, cualquier arista en GT que deje C debe ser a componentes fuertemente conectados que ya han sido visitados. Por lo tanto, ningún vértice

en cualquier componente fuertemente conectado que no sea C será un descendiente de u durante la búsqueda en profundidad de GT. Por lo tanto, los vértices del árbol primero en profundidad en GT que tiene su raíz en u forman exactamente un componente fuertemente conectado, que completa el paso inductivo y la demostración. ■

Aquí hay otra forma de ver cómo funciona la segunda búsqueda en profundidad. Considere el gráfico de componentes .GT/SCC de GT. Si mapeamos cada componente fuertemente conectado visitado en la segunda búsqueda en profundidad a un vértice de .GT/SCC, la segunda búsqueda en profundidad visita los vértices de .GT/SCC en orden inverso a la clasificación topológica. Si invertimos los bordes de .GT/SCC, obtenemos el gráfico ..GT/SCC/T.

Debido a que ..GT/SCC/T D GSCC (vea el ejercicio 22.5-4), la segunda búsqueda en profundidad visita los vértices de GSCC en orden clasificado topológicamente.

### Ejercicios

#### 22.5-1

¿Cómo puede cambiar el número de componentes fuertemente conectados de un gráfico si se agrega una nueva arista?

#### 22.5-2

Muestre cómo funciona el procedimiento COMPONENTES FUERTEMENTE CONECTADOS en el gráfico de la figura 22.6. Específicamente, muestre los tiempos de finalización calculados en la línea 1 y el bosque producido en la línea 3. Suponga que el ciclo de las líneas 5–7 de DFS considera los vértices en orden alfabético y que las listas de adyacencia están en orden alfabético.

#### 22.5-3

El profesor Bacon afirma que el algoritmo para componentes fuertemente conectados sería más simple si utilizara el gráfico original (en lugar de la transposición) en la segunda búsqueda en profundidad y escaneara los vértices en orden creciente de tiempos de finalización . ¿Este algoritmo más simple siempre produce resultados correctos?

#### 22.5-4

Demuestre que para cualquier grafo dirigido G, tenemos ..GT/SCC/T D GSCC. Es decir, la transposición de la gráfica de componentes de GT es la misma que la gráfica de componentes de G.

#### 22.5-5

Proporcione un algoritmo OV CE-tiempo para calcular la gráfica de componentes de una gráfica dirigida GD .V; M/. Asegúrese de que haya como máximo un borde entre dos vértices en el gráfico de componentes que produce su algoritmo.

## 22.5-6

Dado un grafo dirigido  $G = \langle V; E \rangle$ , explique cómo crear otro gráfico  $G_0 = \langle V; E_0 \rangle$  tales que (a)  $G_0$  tiene las mismas componentes fuertemente conectadas que  $G$ , (b)  $G_0$  tiene la misma gráfica de componentes que  $G$ , y (c)  $E_0$  es lo más pequeño posible. Describa un algoritmo rápido para calcular  $G_0$ .

## 22.5-7

Un grafo dirigido  $G = \langle V; E \rangle$  es semiconexo si, para todos los pares de vértices  $u, v \in V$  tenemos  $u \rightarrow v$  o  $v \rightarrow u$ . Proporcione un algoritmo eficiente para determinar si  $G$  es o no semiconexo. Demuestre que su algoritmo es correcto y analice su tiempo de ejecución.

## Problemas

**22-1 Clasificación de bordes por búsqueda primero en anchura** Un bosque primero en profundidad clasifica los bordes de un gráfico en bordes de árbol, atrás, adelante y cruzados. También se puede usar un árbol primero en anchura para clasificar los bordes accesibles desde el origen de la búsqueda en las mismas cuatro categorías.

a. Demuestre que en una búsqueda primero en anchura de un grafo no dirigido, la siguiente propiedad sostienen:

1. No hay bordes traseros ni bordes delanteros.
2. Para cada arista de árbol  $u; v$ , tenemos  $:d(v) - d(u) = 1$ .
3. Para cada arista transversal  $u; v$ , tenemos  $:d(v) - d(u) \geq 2$ .

b. Demuestre que en una búsqueda primero en anchura de un grafo dirigido, las siguientes propiedades sostienen:

1. No hay bordes delanteros.
2. Para cada arista de árbol  $u; v$ , tenemos  $:d(v) - d(u) = 1$ .
3. Para cada arista transversal  $u; v$ , tenemos  $:d(v) - d(u) \geq 1$ .
4. Para cada borde posterior  $u; v$ , tenemos  $0 < :d(v) - d(u)$ .

**22-2 Puntos de articulación, puentes y componentes biconectados** Sean  $G = \langle V; E \rangle$  sea un grafo conexo no dirigido. Un punto de articulación de  $G$  es un vértice cuya eliminación desconecta a  $G$ . Un puente de  $G$  es una arista cuya eliminación desconecta a  $G$ . Un componente biconectado de  $G$  es un conjunto máximo de aristas tal que dos aristas cualesquiera del conjunto se encuentran en un ciclo simple común. La Figura 22.10 ilustra

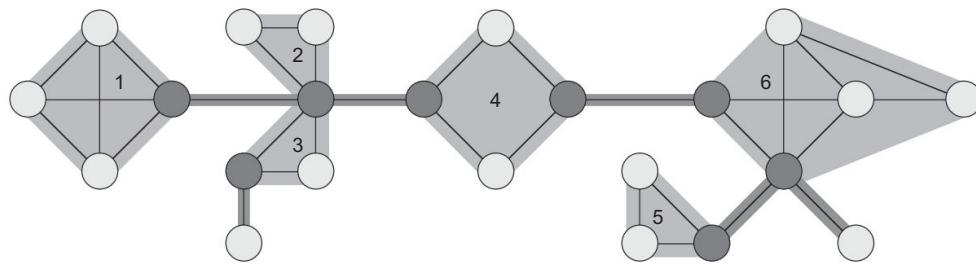


Figura 22.10 Puntos de articulación, puentes y componentes biconectados de un grafo conexo no dirigido para usar en el problema 22-2. Los puntos de articulación son los vértices muy sombreados, los puentes son los bordes muy sombreados y los componentes biconectados son los bordes en las regiones sombreadas, con una numeración bcc mostrada.

estas definiciones. Podemos determinar los puntos de articulación, los puentes y los componentes biconectados mediante la búsqueda en profundidad. Sea  $G = \langle V, E \rangle$  sea un árbol de profundidad primero de  $G$ .

- Demuestre que la raíz de  $G$  es un punto de articulación de  $G$  si y sólo si tiene en menos dos hijos en  $G$ .
- Sea un vértice no raíz de  $G$ . Demuestre que es un punto de articulación de  $G$  si y sólo si tiene un hijo  $s$  tal que no hay un borde posterior desde  $s$  o cualquier descendiente de  $s$  hasta un ancestro propio de  $s$ .

C. Dejar

:

:baja  $D_{\min}(\cdot)$  :d    w:d  $W(u)$ ;  $w$  es un borde posterior para algún descendiente  $u$  de:

Muestre cómo calcular :low para todos los vértices  $V$  en  $O(E)$ /tiempo.

- Muestre cómo calcular todos los puntos de articulación en  $O(E)$ / tiempo.

mi. Demuestre que una arista de  $G$  es un puente si y sólo si no se encuentra sobre ninguna simple ciclo de  $G$

- Muestre cómo calcular todos los puentes de  $G$  en  $O(E)$ /tiempo.

gramo. Demuestre que las componentes biconexas de  $G$  dividen las aristas que no son puente de  $G$ .

- Proporcione un algoritmo  $O(E)$ -tiempo para etiquetar cada arista  $e$  de  $G$  con un entero positivo  $e.bcc$  tal que  $e.bcc = e0.bcc$  si y solo si  $e$  y  $e0$  están en el mismo componente biconectado.

## 22-3 Recorrido de Euler

Euler Recorrido de Euler de un grafo dirigido fuertemente conexo  $G$ .  $V; E$  es un ciclo que atraviesa cada arista de  $G$  exactamente una vez, aunque puede visitar un vértice más de una vez.

- a. Demostrar que  $G$  tiene un recorrido de Euler si y sólo si en grado./  $D$  en grado exterior./ para cada vértice  $2 V$ .
- b. Describa un algoritmo OE/-tiempo para encontrar un recorrido de Euler por  $G$ , si existe. (Pista: Fusionar ciclos disjuntos de aristas.)

## 22-4 Accesibilidad Sea

$G$  . $V; E$  sea un grafo dirigido en el que cada vértice  $u \in V$  esté etiquetado con un único número entero  $L_u$  del conjunto  $f_1; 2; \dots; jV$ . Para cada vértice  $u \in V$  sea  $R_u$  el conjunto de vértices que son alcanzables desde  $u$ . Defina  $\min_u$  como el vértice en  $R_u$  cuya etiqueta es mínima, es decir,  $\min_u$  es el vértice tal que  $L_u \leq L_w \leq R_u$ . Proporcione un algoritmo OV CE/-time que calcule  $\min_u$  para todos los vértices  $u \in V$ .

## Notas del capítulo

Incluso [103] y Tarjan [330] son excelentes referencias para algoritmos gráficos.

La búsqueda primero en amplitud fue descubierta por Moore [260] en el contexto de encontrar caminos a través de laberintos. Lee [226] descubrió de forma independiente el mismo algoritmo en el contexto del enrutamiento de cables en placas de circuitos.

Hopcroft y Tarjan [178] defendieron el uso de la representación de lista de adyacencia sobre la representación de matriz de adyacencia para gráficos dispersos y fueron los primeros en reconocer la importancia algorítmica de la búsqueda en profundidad. La búsqueda en profundidad se ha utilizado ampliamente desde finales de la década de 1950, especialmente en programas de inteligencia artificial.

Tarjan [327] proporcionó un algoritmo de tiempo lineal para encontrar componentes fuertemente conectados. El algoritmo para componentes fuertemente conectados en la Sección 22.5 está adaptado de Aho, Hopcroft y Ullman [6], quienes atribuyen su crédito a SR Kosaraju (inédito) y M. Sharir [314]. Gabow [119] también desarrolló un algoritmo para componentes fuertemente conectados que se basa en ciclos de contratación y utiliza dos pilas para que se ejecute en tiempo lineal. Knuth [209] fue el primero en dar un algoritmo de tiempo lineal para la ordenación topológica.

## 23

## Árboles de expansión mínimos

Los diseños de circuitos electrónicos a menudo necesitan hacer que los pines de varios componentes sean eléctricamente equivalentes al cablearlos juntos. Para interconectar un conjunto de  $n$  pines, podemos usar un arreglo de  $n-1$  cables, cada uno conectando dos pines. De todos estos arreglos, el que utiliza la menor cantidad de alambre suele ser el más deseable.

Podemos modelar este problema de cableado con un grafo  $G = (V, E)$  conectado no dirigido; donde  $V$  es el conjunto de pines,  $E$  es el conjunto de posibles interconexiones entre pares de pines, y para cada arista  $u, v \in E$ , tenemos un peso  $w_{uv}$ ; especificando el costo (cantidad de cable necesario) para conectar  $u$  y  $v$ . Entonces deseamos encontrar un subconjunto acíclico  $T \subseteq E$  que conecte todos los vértices y cuyo peso total

$$\min_{T \subseteq E} \sum_{(u,v) \in T} w_{uv}$$

se minimiza. Dado que  $T$  es acíclico y conecta todos los vértices, debe formar un árbol, al que llamamos árbol de expansión ya que "atrapa" el gráfico  $G$ . Llamamos al problema de la Figura determinando el árbol  $T$  el problema del árbol de expansión mínima.<sup>1</sup> Figura 23.1 muestra un ejemplo de un gráfico conectado y un árbol de expansión mínimo.

En este capítulo, examinaremos dos algoritmos para resolver el problema del árbol de expansión mínimo: el algoritmo de Kruskal y el algoritmo de Prim. Podemos hacer que cada uno de ellos se ejecute fácilmente en el tiempo  $O(|V| \cdot |E| \lg |V|)$  usando montones binarios ordinarios. Mediante el uso de montones de Fibonacci, el algoritmo de Prim se ejecuta en el tiempo  $O(|V|^2 \lg |V|)$ , lo que mejora la implementación del montón binario si  $|V|$  es mucho menor que  $|E|$ .

Los dos algoritmos son algoritmos codiciosos, como se describe en el Capítulo 16. Cada paso de un algoritmo codicioso debe hacer una de varias elecciones posibles. La estrategia codiciosa aboga por hacer la elección que sea mejor en ese momento. Tal estrategia generalmente no garantiza que siempre encontrará soluciones globalmente óptimas.

---

<sup>1</sup>La frase "árbol de expansión mínimo" es una forma abreviada de la frase "árbol de expansión de peso mínimo". No estamos, por ejemplo, minimizando el número de aristas en  $T$  exactamente  $|V|-1$ , ya que todos los árboles de expansión tienen  $|V|-1$  aristas por el Teorema B.2.

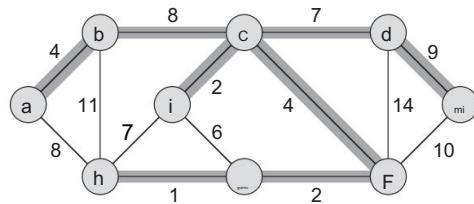


Figura 23.1 Un árbol de expansión mínima para un gráfico conexo. Se muestran los pesos en los bordes y se sombrean los bordes en un árbol de expansión mínima. El peso total del árbol que se muestra es 37. Este árbol de expansión mínimo no es único: quitando el borde .b; c/ y sustituyéndolo por la arista .a; h/ produce otro árbol de expansión con un peso de 37.

a los problemas Sin embargo, para el problema del árbol de expansión mínimo, podemos demostrar que ciertas estrategias codiciosas producen un árbol de expansión con un peso mínimo. Aunque puede leer este capítulo independientemente del Capítulo 16, los métodos codiciosos que se presentan aquí son una aplicación clásica de las nociones teóricas allí presentadas.

La Sección 23.1 introduce un método de árbol de expansión mínimo "genérico" que hace crecer un árbol de expansión agregando un borde a la vez. La sección 23.2 da dos algoritmos que implementan el método genérico. El primer algoritmo, debido a Kruskal, es similar al algoritmo de componentes conectados de la Sección 21.1. El segundo, debido a Prim, se parece al algoritmo de caminos más cortos de Dijkstra (Sección 24.3).

Debido a que un árbol es un tipo de gráfico, para ser precisos debemos definir un árbol en términos no solo de sus bordes, sino también de sus vértices. Aunque este capítulo se centra en los árboles en términos de sus aristas, trabajaremos con el entendimiento de que los vértices de un árbol  $T$  son aquellos sobre los que incide alguna arista de  $T$ .

### 23.1 Crecimiento de un árbol de expansión mínima

Supongamos que tenemos un grafo conexo no dirigido  $G$ ;  $E$  con una función de peso  $w$  !  $R$ , y deseamos encontrar un árbol de expansión mínimo para  $G$ . Los dos algoritmos que consideramos en este capítulo utilizan un enfoque codicioso del problema, aunque difieren en cómo aplican este enfoque.

Esta estrategia codiciosa se captura mediante el siguiente método genérico, que hace crecer el árbol de expansión mínimo un borde a la vez. El método genérico gestiona un conjunto de aristas  $A$ , manteniendo invariante el siguiente bucle:

Antes de cada iteración,  $A$  es un subconjunto de algún árbol de expansión mínimo.

En cada paso, determinamos un borde .u; / que podemos sumar a  $A$  sin violar esta invariante, en el sentido de que  $A \cup u$ ; /g también es un subconjunto de una extensión mínima

árbol. A tal arista la llamamos arista segura para A, ya que podemos sumarla con seguridad a A manteniendo la invariante.

GENÉRICO-MST.G; con 1 dC;

2 mientras que

A no forma un árbol de expansión 3

encontrar un borde .u; / eso es seguro para AADA

[ fu; /g 4 5 vuelta A

Usamos el bucle invariante de la siguiente manera:

Inicialización: después de la línea 1, el conjunto A satisface trivialmente la invariante del bucle.

Mantenimiento: el ciclo en las líneas 2 a 4 mantiene el invariante al agregar solo seguridad  
bordes

Terminación: todas las aristas agregadas a A están en un árbol de expansión mínimo, por lo que el conjunto A  
devuelto en la línea 5 debe ser un árbol de expansión mínimo.

La parte complicada es, por supuesto, encontrar un borde seguro en la línea 3. Uno debe existir, ya que  
cuando se ejecuta la línea 3, el invariante dicta que hay un árbol de expansión T tal que AT . Dentro del cuerpo  
del bucle while , A debe ser un subconjunto propio de T y, por lo tanto, debe haber una arista .u; / 2 T tal que .u; /  
62 A y .u; / es seguro para A.

En el resto de esta sección, proporcionamos una regla (Teorema 23.1) para reconocer bordes seguros. La  
siguiente sección describe dos algoritmos que usan esta regla para encontrar bordes seguros de manera eficiente.

Primero necesitamos algunas definiciones. Un corte .S; VS / de un grafo no dirigido GD .V; E/ es una partición  
de V . La Figura 23.2 ilustra esta noción. Decimos que una arista .u; / 2 E cruza el corte .S; VS / si uno de sus  
extremos está en S y el otro en V S. Decimos que un corte respeta un conjunto A de aristas si ninguna arista en  
A cruza el corte. Un borde es un borde ligero que cruza un corte si su peso es el mínimo de cualquier borde que  
cruce el corte. Tenga en cuenta que puede haber más de un borde ligero que cruce un corte en el caso de las  
corbatas. Más generalmente, decimos que un borde es un borde ligero que satisface una propiedad dada si su  
peso es el mínimo de cualquier borde que satisfaga la propiedad.

Nuestra regla para reconocer bordes seguros viene dada por el siguiente teorema.

### Teorema 23.1

Sea GD .V; E/ sea un grafo conexo, no dirigido, con una función de ponderación de valor real  
ción w definida en E. Sea A un subconjunto de E que está incluido en algún mínimo  
árbol de expansión para G, sea .S; VS / sea cualquier corte de G que respete A, y sea .u; / ser un cruce de borde  
ligero .S; VS/. Entonces, borde .u; / es seguro para A.

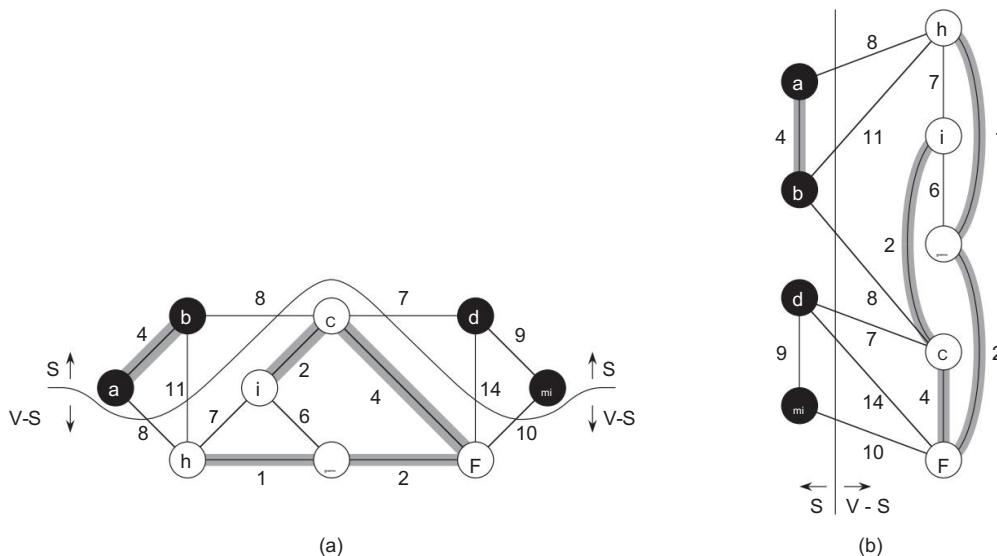


Figura 23.2 Dos formas de ver un corte  $.S; VS /$  del gráfico de la figura 23.1. (a) Los vértices negros están en el conjunto  $S$  y los vértices blancos están en  $V_S$ . Las aristas que cruzan el corte son las que conectan los vértices blancos con los vértices negros. El borde  $.d; c/$  es el único borde ligero que cruza el corte. Un subconjunto  $A$  de los bordes está sombreado; tenga en cuenta que el corte  $.S; VS /$  respeta  $A$ , ya que ninguna arista de  $A$  cruza el corte. (b) El mismo gráfico con los vértices en el conjunto  $S$  a la izquierda y los vértices en el conjunto  $VS$  a la derecha. Una arista cruza el corte si conecta un vértice de la izquierda con un vértice de la derecha.

Demostración Sea  $T$  un árbol de expansión mínima que incluye  $A$ , y suponga que  $T$  no contiene la arista ligera  $.u; /$ , ya que si lo hace, hemos terminado. Construiremos otra técnica de cortar y pegar del árbol de  $T^0$  eso incluye  $A$  [fu; /g usando un expansión mínima  $T$ , demostrando así que  $.u; /$  es un borde seguro para  $A$ .

El borde  $.u; /$  forma un ciclo con los bordes en el camino simple  $p$  desde  $u$  hasta en  $T$  como ilustra la figura 23.3. Como  $u$  y están en lados opuestos del corte  $.S; VS /$ , al menos una arista en  $T$  se encuentra en el camino simple  $p$  y también cruza el corte. Sea  $.x; y /$  ser tal borde. El borde  $.x; y/$  no está en  $A$ , porque el corte respeta a  $A$ . Como  $.x; y/$  está en el único camino simple de  $u$  a en  $T$  ing  $.x; y/$  divide a  $T$ , eliminar en dos componentes. Agregando  $.u; /$  los vuelve a conectar para formar un nuevo árbol de expansión  $T^0 DT fx; y/g [fu; /gramo.$

A continuación mostramos que  $T^0$  es un árbol de expansión mínima. Dado que  $.u; /$  es un cruce de borde ligero  $.S; VS / y .x; y/$  también cruza este corte,  $wu; / wx; y/$ . Por lo tanto, peso $^0/D wT / wx; y/ C wu; / wT / :$

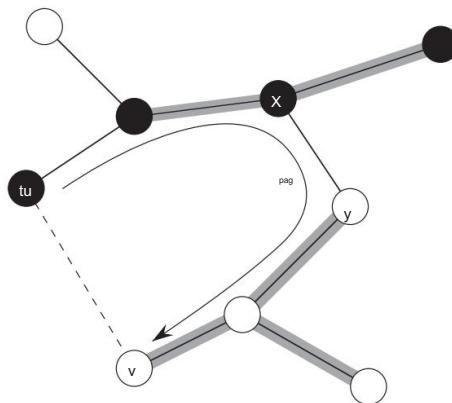


Figura 23.3 La demostración del Teorema 23.1. Los vértices negros están en  $S$  y los vértices blancos están en  $V \setminus S$ . Se muestran las aristas en el árbol de expansión mínima  $T$ , pero no las aristas en el gráfico  $G$ . Los bordes en  $A$  están sombreados y  $.u; /$  es un borde ligero que cruza el corte  $.S; V \setminus S/$ . El borde  $.x; y/$  es una arista en el único camino simple  $p$  desde  $u$  hasta  $x$  en  $T$ . Para formar un árbol generador mínimo,  $T$  contiene  $.u; /$ , quita el  $.x; y/$  de  $T$  y agrega el borde  $.u; /$ .

Pero  $T$  es un árbol de expansión mínima, por lo que  $w_T / w_T$  también es un  $^0$ ; por lo tanto,  $T^0$  debe ser un árbol de expansión mínima.

Queda por demostrar que  $.u; /$  es en realidad un borde seguro para  $A$ . Tenemos  $AT$  es un  $^0$ , desde  $AT$  y  $.x; y/ \not\in A$ ; así,  $A [fu; /g T$  árbol de expansión mínima,  $.u; /$ . En consecuencia, dado que  $T^0$  es seguro para  $A$ . ■

El teorema 23.1 nos da una mejor comprensión del funcionamiento del método GENERIC MST en un grafo conectado  $G(V; E)$ . A medida que avanza el método, el conjunto  $A$  siempre es acíclico; de lo contrario, un árbol de expansión mínima que incluya a  $A$  contendría un ciclo, lo cual es una contradicción. En cualquier punto de la ejecución, el gráfico  $GA = (V; A)$  es un bosque, y cada uno de los componentes conectados de  $GA$  es un árbol.

(Algunos de los árboles pueden contener solo un vértice, como es el caso, por ejemplo, cuando el método comienza:  $A$  está vacío y el bosque contiene  $j$  árboles, uno para cada vértice.) Además, cualquier borde seguro  $.u; /$  para  $A$  conecta distintos componentes de  $GA$ , ya que  $A [fu; /g$  debe ser acíclico.

El bucle while de las líneas 2 a 4 de GENERIC-MST ejecuta  $j$  veces porque  $1$  veces porque una de las aristas  $j$  de un árbol de expansión mínima en cada iteración.

Inicialmente, cuando  $AD = \emptyset$ , hay  $j$  árboles en  $GA$ , y cada iteración reduce ese número en 1. Cuando el bosque contiene un solo árbol, el método termina.

Los dos algoritmos de la Sección 23.2 utilizan el siguiente corolario del Teorema 23.1.

**Corolario 23.2**

Sea  $G = (V, E)$ ; Sea  $E'$  un grafo conexo, no dirigido, con una función de ponderación de valor real  $w$  definida en  $E$ . Sea  $A$  un subconjunto de  $E$  que está incluido en algún árbol generador mínimo para  $G$ , y sea  $C = (V, E'')$  / ser un componente conexo (árbol) en el bosque  $G - A$ . Si  $u \in V(C) \setminus A$ ; / es un borde claro que conecta  $C$  con algún otro componente en  $G - A$ , luego  $u$ ; / es seguro para  $A$ .

Prueba El corte  $V \setminus C$ ; / respeta  $A$ , y  $u$ ; / es un borde ligero para este corte.

Por lo tanto,  $u$ ; / es seguro para  $A$ . ■

**Ejercicios****23.1-1**

Sea  $u$ ; / sea una arista de peso mínimo en un grafo conexo  $G$ . Demuestre que  $u$ ; / pertenece a algún árbol de expansión mínimo de  $G$ .

**23.1-2**

El profesor Sabatier conjectura el siguiente recíproco del teorema 23.1. Sea  $G = (V, E)$ ; Sea  $E'$  un grafo conexo, no dirigido, con una función de ponderación de valor real  $w$  definida en  $E$ . Sea  $A$  un subconjunto de  $E$  que está incluido en algún árbol generador mínimo para  $G$ , sea  $S = V \setminus A$ ; / sea cualquier corte de  $G$  que respete  $A$ , y sea  $u$ ; / ser un borde seguro para un cruce  $S$ ;  $V \setminus S$ . El  $u$ ; / es un borde ligero para el corte. Demuestra que la conjectura del profesor es incorrecta dando un contraejemplo.

**23.1-3**

Muestre que si una arista  $u$ ; / está contenido en algún árbol de expansión mínimo, entonces es un borde ligero que cruza algún corte del gráfico.

**23.1-4**

Da un ejemplo simple de un grafo conexo tal que el conjunto de aristas  $u$ ; /  $W$  existe un corte  $S$ ;  $V \setminus S$  / tal que  $u$ ; / es un cruce de borde ligero  $S$ ;  $V \setminus S$  / no forma un árbol de expansión mínimo.

**23.1-5**

Sea  $e$  una arista de peso máximo en algún ciclo del grafo conexo  $G = (V, E)$ .

Demuestre que existe un árbol generador mínimo de  $G - e$ ; / es un árbol de expansión mínimo de  $G$ . Es decir, hay un árbol de expansión mínimo de  $G$  que no incluye a  $e$ .

## 23.1-6

Muestre que un grafo tiene un árbol de expansión mínimo único si, para cada corte del grafo, hay un borde claro único que cruza el corte. Demuestre que lo contrario no es cierto dando un contrajeemplo.

## 23.1-7

Argumente que si todos los pesos de las aristas de un gráfico son positivos, entonces cualquier subconjunto de aristas que conecte todos los vértices y tenga un peso total mínimo debe ser un árbol. Dé un ejemplo para mostrar que no se sigue la misma conclusión si permitimos que algunos pesos sean no positivos.

## 23.1-8

Sea  $T$  un árbol generador mínimo de un grafo  $G$ , y sea  $L$  la lista ordenada de los pesos de las aristas de  $T$ . Muestre que para cualquier otro árbol generador mínimo  $T'$  de  $G$ , la lista  $L$  es también la lista ordenada de pesos de arista de  $T'$ .

## 23.1-9

Sea  $T$  un árbol generador mínimo de un grafo  $G$ , y sea  $V$  un subconjunto de  $V$ . Sea  $T'$  un subconjunto de  $T$  que es el subgrafo de  $T$  inducido por  $V$ . Sea  $G'$  el subgrafo de  $G$  que es un árbol inducido por  $V$ . Demuestre que si  $T'$  es conexo, entonces  $T'$  es un generador mínimo de  $G'$ .

## 23.1-10

Dado un grafo  $G$  y un árbol generador mínimo  $T$ , suponga que disminuimos el peso de una de las aristas en  $T$ . Muestre que  $T$  sigue siendo un árbol de expansión mínimo para  $G$ . Más formalmente, sea  $T'$  un árbol de expansión mínimo para  $G$  con pesos de borde dados por la función de peso  $w$ . Elija un borde  $x; y$  y un número positivo  $k$ , y defina la función de peso  $w_0$  por

$$\text{si } tu; / \neq x; y; wx; y/k$$

$$w_0(u; / D(wu; / \text{si } u; / D.x; y) :$$

Muestre que  $T'$  es un árbol de expansión mínimo para  $G$  con pesos de borde dados por  $w_0$ .

## 23.1-11 ?

Dado un grafo  $G$  y un árbol generador mínimo  $T$ , supongamos que disminuimos el peso de una de las aristas que no está en  $T$ . Proporcione un algoritmo para encontrar el árbol de expansión mínimo en el gráfico modificado.

## 23.2 Los algoritmos de Kruskal y Prim

Los dos algoritmos de árbol de expansión mínimo descritos en esta sección elaboran el método genérico. Cada uno usa una regla específica para determinar un borde seguro en la línea 3 de GENERIC-MST. En el algoritmo de Kruskal, el conjunto A es un bosque cuyos vértices son todos los del grafo dado. El borde seguro agregado a A es siempre un borde de menor peso en el gráfico que conecta dos componentes distintos. En el algoritmo de Prim, el conjunto A forma un solo árbol. El borde seguro agregado a A es siempre un borde de menor peso que conecta el árbol con un vértice que no está en el árbol.

### Algoritmo de Kruskal

El algoritmo de Kruskal encuentra un borde seguro para agregar al bosque en crecimiento al encontrar, de todos los bordes que conectan dos árboles en el bosque, un borde  $.u; /$  de menor peso. Sean C1 y C2 los dos árboles que están conectados por  $.u; /$ . Dado que  $.u; /$  debe ser una arista clara que conecte C1 con algún otro árbol, el Corolario 23.2 implica que  $.u; /$  es un borde seguro para C1. El algoritmo de Kruskal califica como un algoritmo codicioso porque en cada paso agrega al bosque un borde de menor peso posible.

Nuestra implementación del algoritmo de Kruskal es como el algoritmo para calcular componentes conectados de la Sección 21.1. Utiliza una estructura de datos de conjuntos disjuntos para mantener varios conjuntos disjuntos de elementos. Cada conjunto contiene los vértices de un árbol del bosque actual. La operación FIND-SET. $u/$  devuelve un elemento representativo del conjunto que contiene u. Por lo tanto, podemos determinar si dos vértices u y pertenecen al mismo árbol probando si FIND-SET. $u/$  es igual a FIND-SET. $/$ . Para combinar árboles, el algoritmo de Kruskal llama al procedimiento UNION .

```
MST-KRUSKAL.G; con 1
dC; 2 para
cada vértice 2 G:V 3 MAKE-SET./
4 ordenar las aristas de
G:E en orden no decreciente por peso w 5 para cada arista .u; / 2 G:E,
tomado en orden no decreciente por peso 6 if FIND-SET.u/ ≠ FIND-SET./ 7 ADA [ fu; /
g 8 UNIÓN.u; / 9 vuelta A
```

La figura 23.4 muestra cómo funciona el algoritmo de Kruskal. Las líneas 1 a 3 inicializan el conjunto A en el conjunto vacío y crean árboles  $jV j$ , uno que contiene cada vértice. El ciclo for en las líneas 5 a 8 examina los bordes en orden de peso, de menor a mayor. El lazo

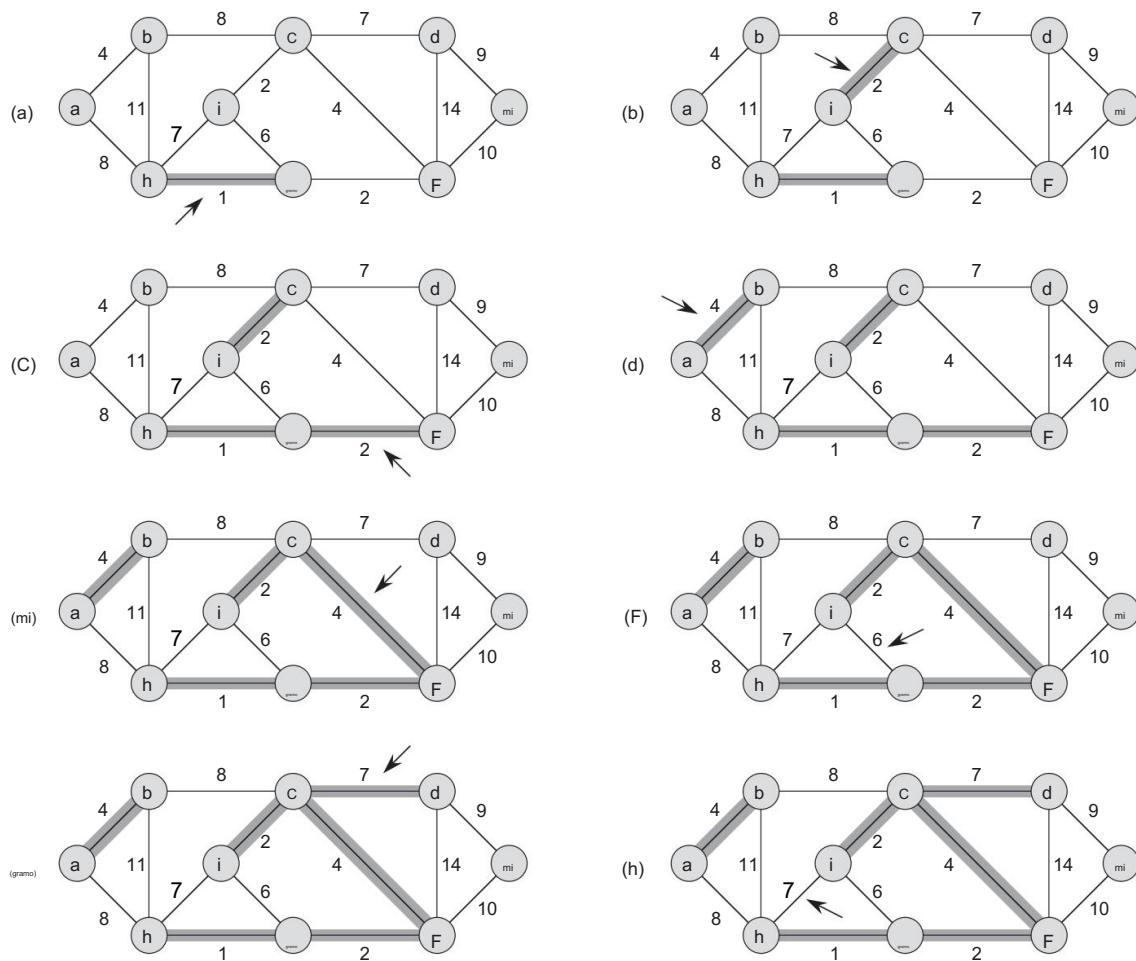


Figura 23.4 La ejecución del algoritmo de Kruskal en el gráfico de la Figura 23.1. Los bordes sombreados pertenecen al bosque A que se está cultivando. El algoritmo considera cada borde ordenado por peso. Una flecha apunta al borde bajo consideración en cada paso del algoritmo. Si el borde une dos árboles distintos en el bosque, se agrega al bosque, fusionando así los dos árboles.

cheques, para cada borde  $.u; /$ , si los extremos  $u$  y  $/$  pertenecen al mismo árbol. Si lo hacen, entonces el borde  $.u; /$  no se puede agregar al bosque sin crear un ciclo, y el borde se descarta. De lo contrario, los dos vértices pertenecen a árboles diferentes. En este caso, la línea 7 agrega el borde  $.u; /$  a  $A$ , y la línea 8 une los vértices de los dos árboles.

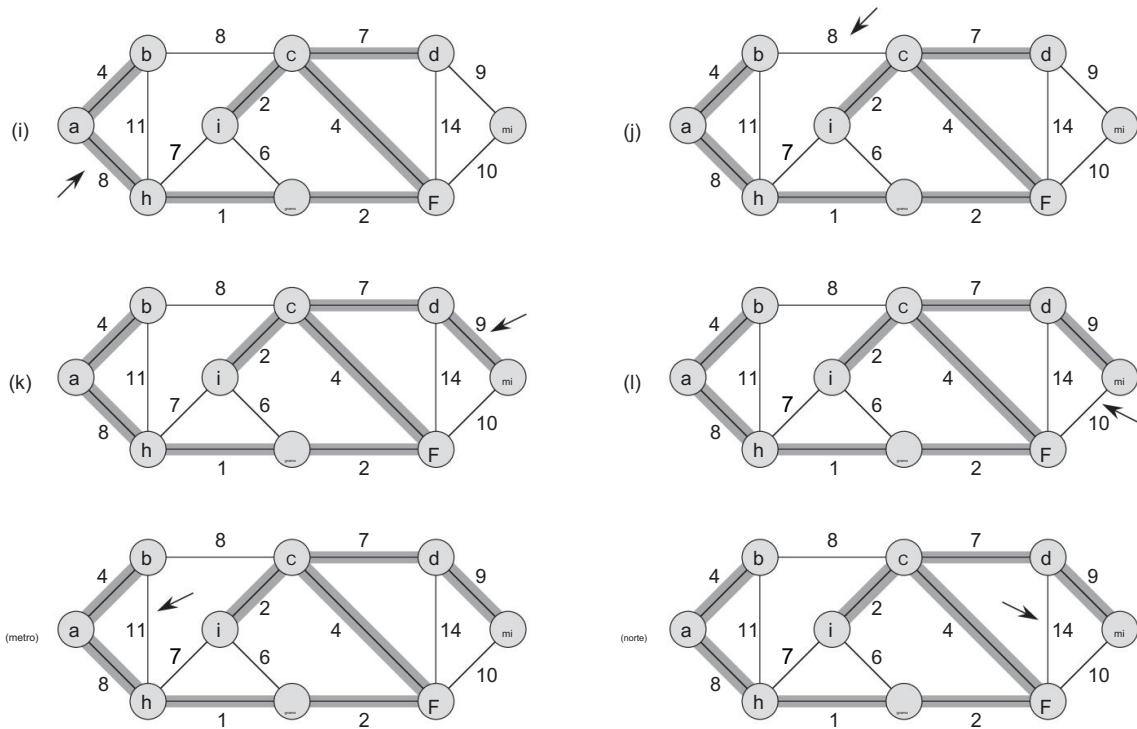


Figura 23.4, continuación Pasos adicionales en la ejecución del algoritmo de Kruskal.

El tiempo de ejecución del algoritmo de Kruskal para un grafo  $G = \langle V, E \rangle$  depende de cómo implementemos la estructura de datos del conjunto disjunto. Suponemos que usamos la implementación de bosque de conjunto disjunto de la Sección 21.3 con las heurísticas de unión por rango y compresión de ruta, ya que es la implementación asintóticamente más rápida conocida. Inicializar el conjunto A en la línea 1 toma  $O(1)$  tiempo, y el tiempo para ordenar los bordes en la línea 4 es  $O(|E| \lg |E|)$ . (En un momento daremos cuenta del costo de las operaciones  $\text{MAKE-SET}$  en el bucle `for` de las líneas 2 y 3). El bucle `for` de las líneas 5 a 8 realiza operaciones  $O(|E| / |V|)$  FIND-SET y UNION en el bucle disjunto establecer bosque. Junto con las operaciones  $\text{MAKE-SET}$ , estas toman un total de  $O(|V| \cdot \alpha(|V|))$  tiempo, donde  $\alpha$  es la función de crecimiento muy lento definida en la Sección 21.4. Como asumimos que  $G$  es conexo, tenemos  $|E| \geq |V| - 1$ , por lo que las operaciones con conjuntos disjuntos toman  $O(|V| \cdot \alpha(|V|))$  tiempo. Además, dado que  $\alpha(|V|) \leq O(\lg |V|)$ , el tiempo de ejecución total del algoritmo de Kruskal es  $O(|E| \lg |E|)$ . Observando que  $|E| < |V|^2$ , tenemos  $\lg |E| \leq O(\lg |V|)$ , y por tanto podemos reformular el tiempo de ejecución del algoritmo de Kruskal como  $O(|V| \cdot \alpha(|V|))$ .

### algoritmo de Prim

Al igual que el algoritmo de Kruskal, el algoritmo de Prim es un caso especial del método genérico del árbol de expansión mínimo de la Sección 23.1. El algoritmo de Prim funciona de manera muy similar al algoritmo de Dijkstra para encontrar los caminos más cortos en un gráfico, que veremos en la Sección 24.3. El algoritmo de Prim tiene la propiedad de que las aristas del conjunto A siempre forman un solo árbol. Como muestra la Figura 23.5, el árbol parte de un vértice raíz arbitrario  $r$  y crece hasta que el árbol abarca todos los vértices. Cada paso agrega al árbol A una arista ligera que conecta a A con un vértice aislado, uno en el que no incide ninguna arista de A. Por el Corolario 23.2, esta regla agrega solo aristas que son seguras para A; por lo tanto, cuando el algoritmo termina, los bordes en A forman un árbol de expansión mínimo. Esta estrategia califica como codiciosa ya que en cada paso agrega al árbol una arista que aporta la mínima cantidad posible al peso del árbol.

Para implementar el algoritmo de Prim de manera eficiente, necesitamos una forma rápida de seleccionar una nueva arista para agregar al árbol formado por las aristas en A. En el pseudocódigo a continuación, el grafo conectado G y la raíz  $r$  del árbol de expansión mínimo a ser crecido son entradas al algoritmo. Durante la ejecución del algoritmo, todos los vértices que no están en el árbol residen en una cola Q de prioridad mínima , el atributo :key es el peso mínimo de cualquier borde que conecte basada en un atributo clave . Para cada vértice a un vértice en el árbol; por convención, :key D 1 si no existe tal borde. El atributo nombría al padre del árbol. El algoritmo mantiene implícitamente el conjunto A de GENERIC-MST como

AD f.; ./ W 2 V frg Qg :

Cuando el algoritmo termina, la cola de prioridad mínima Q está vacía; el árbol de expansión mínimo A para G es por lo tanto

AD f.; ./ W 2 V frgg :

MST-PRIM.G; w; r/

1 para cada u 2 G:V 2 3 4

u: tecla D 1  
tu: D NIL

r:key D 0 5

QDG:V 6 mientras

Q  $\neq$ ; 7 u D EXTRACT-

MIN.Q/ 8 por cada 2 G:AdjEu 9 si 2 Q y  
wu; / < :tecla 10

tu

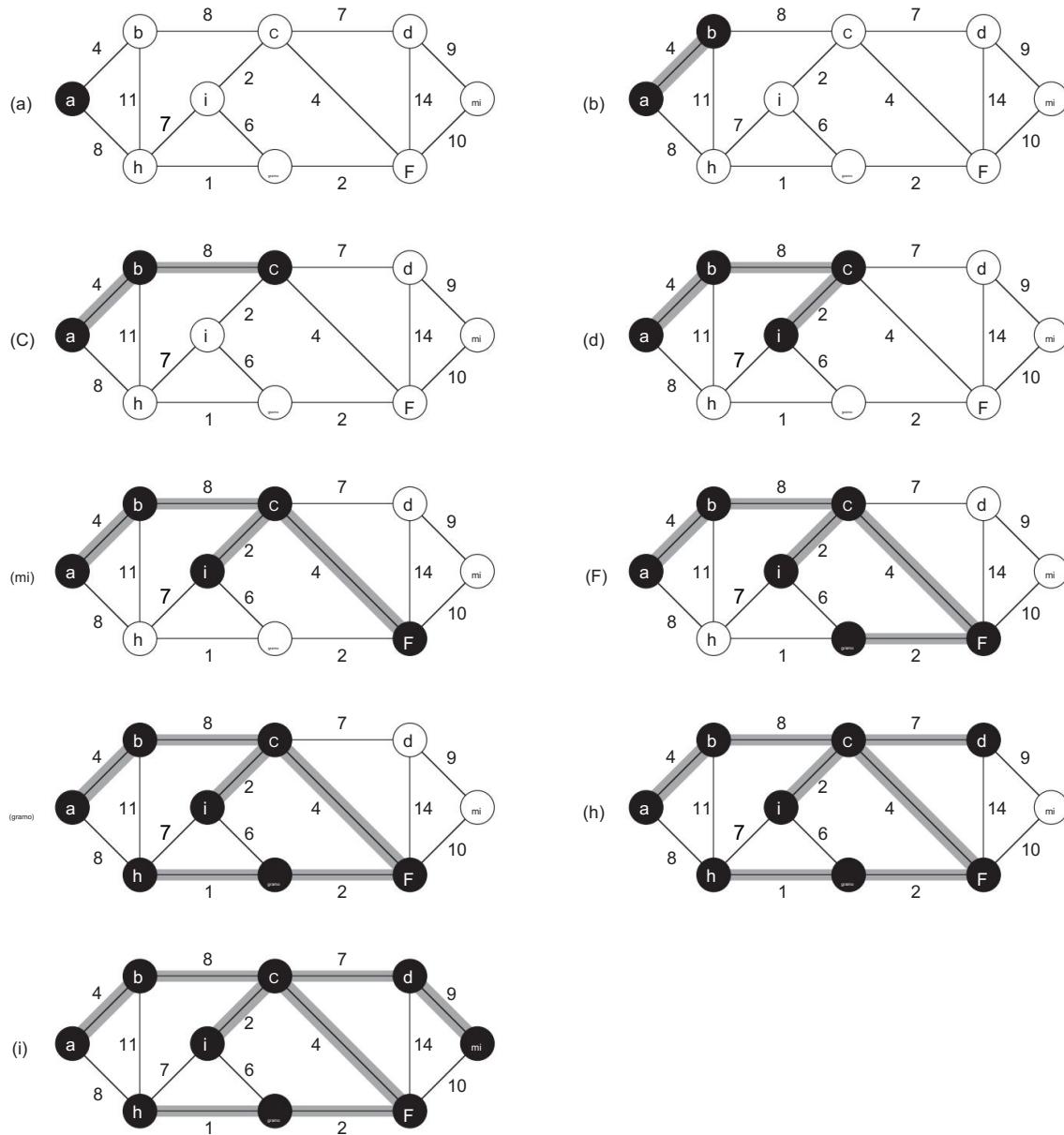


Figura 23.5 La ejecución del algoritmo de Prim en el gráfico de la Figura 23.1. El vértice de la raíz es a. Los bordes sombreados están en el árbol que se está cultivando y los vértices negros están en el árbol. En cada paso del algoritmo, los vértices del árbol determinan un corte del gráfico y se agrega al árbol un borde claro que cruza el corte. En el segundo paso, por ejemplo, el algoritmo tiene la opción de agregar el borde .b; c/ o arista .a; h/ al árbol ya que ambos son bordes ligeros cruzando el corte.

La figura 23.5 muestra cómo funciona el algoritmo de Prim. Las líneas 1 a 5 establecen la clave de cada vértice en 1 (excepto para la raíz  $r$ , cuya clave se establece en 0 para que sea el primer vértice procesado), establezca el padre de cada vértice en NIL e inicialice la prioridad mínima cola  $Q$  para contener todos los vértices. El algoritmo mantiene el siguiente ciclo de tres partes invariante:

Antes de cada iteración del bucle while de las líneas 6 a 11,

1. AD  $f.; :/ W 2 V frg Qg.$
2. Los vértices ya colocados en el árbol de expansión mínimo son los de  $V Q.$
3. Para todos los vértices  $2 Q$ , si  $: \neq NIL$ , entonces  $:key < 1$  y  $:key$  es el peso de una arista ligera  $; :/$  conectando a algún vértice ya colocado en el árbol de expansión mínima.

La línea 7 identifica un vértice  $u$   $2 Q$  incidente en una arista clara que cruza el corte  $.VQ;Q/$  (a excepción de la primera iteración, en la que  $u D r$  debido a la línea 4).

Quitando  $u$  del conjunto  $Q$  lo agrega al conjunto  $VQ$  de vértices en el árbol, sumando así  $.u;$   $u:/$  a  $A.$  El bucle for de las líneas 8 a 11 actualiza la clave y los atributos de cada vértice adyacente a  $u$  pero no en el árbol, manteniendo así invariable la tercera parte del bucle.

El tiempo de ejecución del algoritmo de Prim depende de cómo implementemos la cola de prioridad mínima  $Q$ . Si implementamos  $Q$  como un montón mínimo binario (vea el Capítulo 6), podemos usar el procedimiento BUILD-MIN-HEAP para realizar las líneas 1–5 en  $O(V)$  tiempo. El cuerpo del ciclo while ejecuta  $jV$  veces, y dado que cada operación EXTRACT-MIN toma  $O(\lg V / \text{time})$ , el tiempo total para todas las llamadas a EXTRACT-MIN es  $O(V \lg V / \text{time})$ .

El bucle for de las líneas 8 a 11 ejecuta  $O(E)$  veces por completo, ya que la suma de las longitudes de todas las listas de adyacencia es  $2 jE$ . Dentro del ciclo for , podemos implementar la prueba de membresía en  $Q$  en la línea 9 en tiempo constante manteniendo un bit para cada vértice que indica si está o no en  $Q$ , y actualizando el bit cuando se elimina el vértice de  $Q.$  La asignación en la línea 11 implica una operación implícita de DECREASE-KEY en el montón mínimo, que un montón mínimo binario admite en  $O(\lg V / \text{time})$ . Por lo tanto, el tiempo total para el algoritmo de Prim es  $O(V \lg V + E \lg V / \text{time})$ , que es asintóticamente el mismo que para nuestra implementación del algoritmo de Kruskal.

Podemos mejorar el tiempo de ejecución asintótico del algoritmo de Prim usando montones de Fibonacci. El capítulo 19 muestra que si un montón de Fibonacci contiene  $jV$  elementos, una operación EXTRACT-MIN toma  $O(1 / \text{time})$  amortizado y una operación DECREASE-KEY (para implementar la línea 11) toma  $O(1 / \text{time})$  amortizado. Por lo tanto, si usamos un montón de Fibonacci para implementar la cola de prioridad mínima  $Q$ , el tiempo de ejecución del algoritmo de Prim mejora a  $O(E \lg V / \text{time})$ .

### Ejercicios

#### 23.2-1

El algoritmo de Kruskal puede devolver diferentes árboles de expansión para el mismo grafo de entrada  $G$ , dependiendo de cómo rompa los lazos cuando los bordes se clasifican en orden. Muestre que para cada árbol de expansión mínimo  $T$  de  $G$ , hay una forma de ordenar los bordes de  $G$  en el algoritmo de Kruskal de modo que el algoritmo devuelva  $T$ .

#### 23.2-2

Supongamos que representamos el grafo  $G$   $.V; E/$  como matriz de adyacencia. Proporcione una implementación simple del algoritmo de Prim para este caso que se ejecuta en el tiempo  $O(V^2)$ .

#### 23.2-3

Para un grafo disperso  $G$   $.V; E/$ , donde  $|E| \leq |V|$ , ¿la implementación del algoritmo de Prim con un montón de Fibonacci es asintóticamente más rápida que la implementación del montón binario? ¿Qué pasa con un gráfico denso, donde  $|E| \geq |V|^2$ ? ¿Cómo deben estar relacionados los tamaños  $|E|$  y  $|V|$  para que la implementación del montón de Fibonacci sea asintóticamente más rápida que la implementación del montón binario?

#### 23.2-4

Suponga que todos los pesos de los bordes en un gráfico son números enteros en el rango de 1 a  $|V|$ . ¿Qué tan rápido puedes hacer que se ejecute el algoritmo de Kruskal? ¿Qué pasa si los pesos de los bordes son números enteros en el rango de 1 a  $W$  para alguna constante  $W$ ?

#### 23.2-5

Suponga que todos los pesos de los bordes en un gráfico son números enteros en el rango de 1 a  $|V|$ . ¿Qué tan rápido puedes hacer que se ejecute el algoritmo de Prim? ¿Qué pasa si los pesos de los bordes son números enteros en el rango de 1 a  $W$  para alguna constante  $W$ ?

#### 23.2-6 ?

Suponga que los pesos de los bordes en un gráfico se distribuyen uniformemente sobre el intervalo semiabierto  $[0; 1]$ . ¿Qué algoritmo, el de Kruskal o el de Prim, puedes hacer que corra más rápido?

#### 23.2-7 ?

Suponga que un gráfico  $G$  tiene un árbol de expansión mínimo ya calculado. ¿Qué tan rápido podemos actualizar el árbol de expansión mínimo si agregamos un nuevo vértice y bordes incidentes a  $G$ ?

#### 23.2-8

El profesor Borden propone un nuevo algoritmo divide y vencerás para calcular árboles de expansión mínimos, que es el siguiente. Dado un grafo  $G$   $.V; E/$ , dividir el conjunto  $V$  de vértices en dos conjuntos  $V_1$  y  $V_2$  tales que  $|V_1|$  y  $|V_2|$  difieren

por lo sumo 1. Sea  $E_1$  el conjunto de aristas que inciden sólo en los vértices de  $V_1$ , y sea  $E_2$  el conjunto de aristas que inciden sólo en los vértices de  $V_2$ . Resuelva recursivamente un problema de árbol de expansión mínimo en cada uno de los dos subgrafos  $G_1 \subset V_1; E_1$  y  $G_2 \subset V_2; E_2$ . Finalmente, seleccione el borde de peso mínimo en  $E$  que cruza el corte  $.V_1; V_2/$ , y use este borde para unir los dos árboles de expansión mínimos resultantes en un solo árbol de expansión.

Argumente que el algoritmo calcula correctamente un árbol de expansión mínimo de  $G$  o proporcione un ejemplo en el que el algoritmo falla.

## Problemas

23-1 Segundo mejor árbol de expansión mínimo Sea  $GD .V; E/$  sea un grafo conexo no dirigido cuya función de ponderación sea  $w : WE \rightarrow R$ , y suponga que  $jEj \leq V$  y todos los pesos de los bordes son distintos.

Definimos un segundo mejor árbol de expansión mínimo de la siguiente manera. Sea  $T$  el conjunto de todos los árboles de expansión de  $G$ , y sea  $T$  un árbol de expansión mínimo de  $G$ . Entonces, el segundo mejor árbol de expansión mínimo es un árbol de expansión  $T$  tal que  $w_T / D \min_{T' \in T} w_{T'}$ .

- Muestre que el árbol de expansión mínimo es único, pero que el segundo mejor mini árbol de expansión de mamá no necesita ser único.
- Sea  $T$  el árbol generador mínimo de  $G$ . Demuestre que  $G$  contiene aristas  $.u; v/$  tales que  $w_{uv} \geq w_{T_{uv}}$  tal que  $T_{uv} \subset G$  es el segundo mejor árbol generador mínimo de  $G$ .

C. Sea  $T$  un árbol generador de  $G$  y, para cualesquiera dos vértices  $u; v \in V$ , dejar  $\max_{T' \in T} w_{uv}$ ; denote un borde de peso máximo en el único camino simple entre  $u$  y  $v$  en  $T$  todo  $u; v \in V$ . Describa un algoritmo OV  $2/$ -tiempo que, dado  $T$ , calcule  $\max_{T' \in T} w_{uv}$  para todos los pares  $u; v \in V$ .

- Proporcione un algoritmo eficiente para calcular el segundo mejor árbol de expansión mínimo de  $G$ .

23-2 Árbol de expansión mínimo en grafos dispersos Para un grafo conexo muy disperso  $GD .V; E/$ , podemos mejorar aún más el OE CV lg  $V/tiempo$  de ejecución del algoritmo de Prim con montones de Fibonacci preprocesando  $G$  para disminuir el número de vértices antes de ejecutar el algoritmo de Prim. En particular, elegimos, para cada vértice  $u$ , la arista de peso mínimo  $.u; v/$  incidente en  $u$ , y ponemos  $.u; v/$  en el árbol de expansión mínimo en construcción. Nosotros

luego contraer todos los bordes elegidos (ver Sección B.4). En lugar de contraer estos bordes uno a la vez, primero identificamos conjuntos de vértices que se unen en el mismo vértice nuevo. Luego, creamos el gráfico que habría resultado de contraer estos bordes uno a la vez, pero lo hacemos "renombrando" los bordes de acuerdo con los conjuntos en los que se colocaron sus puntos finales. Varios bordes del gráfico original se pueden renombrar igual que los demás. En tal caso, sólo resulta un borde, y su peso es el mínimo de los pesos de los bordes originales correspondientes.

Inicialmente, establecemos que el árbol de expansión mínimo T que se está construyendo esté vacío, y para cada borde .u; / 2 E, inicializamos los atributos .u; /:orig D .u; / y tú; /:c D wu; /. Usamos el atributo orig para hacer referencia al borde del gráfico inicial que está asociado con un borde en el gráfico contraído. El atributo c tiene el peso de una arista y, a medida que se contraen las aristas, lo actualizamos de acuerdo con el esquema anterior para elegir los pesos de las aristas. El procedimiento MST-REDUCE toma las entradas G y T y devuelve un gráfico contraído G0 con atributos actualizados orig0 y c0 . El procedimiento también acumula aristas de G en el árbol de expansión mínima T

#### MST-REDUCIR.G; T/

```

1 por cada 2 G:V
2           :marca D FALSO
3           MAKE-SET./
4 por cada u 2 G:V
5           si u:marcar == FALSO
6           elegir 2 G:AdjŒeu tal que .u; /:c se minimiza UNION.u; /
7           TDT [fu; /:origg
           u:marca D :marca D
8 9           VERDADERO 10 G0 :V D
fFIND-SET./ W 2 G:Vg 11 G0 :E D ; 12 por
cada .x; y/ 2 G:E
13 u D ENCONTRAR-SET.x/
14 D ENCONTRAR-SET.y/ 15
si .u; / 62 G0 :E 16 G0 :E D
G0 :E [ fu; /Gu; /:orig0 D .x;
y/:origen 17 18 .u; /:c0 D .x; y/:c 19 si
no .x; y/:c < .u; //c0 20

21           .u; /:orig0 D .x;
           y/:origen .u; /:c0 D .x;
y/:c 22 construir listas de adyacencia G0 :Adj
para G0 23 devolver G0 y T

```

- a. Sea  $T$  el conjunto de aristas devuelto por MST-REDUCE, y sea  $A$  el árbol generador mínimo del grafo  $G_0$  formado por la llamada MST-PRIM. $G_0 ; c_0 ; r$ , donde  $c_0$  es el atributo de peso en los bordes de  $G_0 : E$  y  $r$  es cualquier vértice en  $G_0 : V$ . Demuestre que  $T [fx; y/: orig0 W .x; y/ 2 Ag]$  es un árbol generador mínimo de  $G$ .
- b. Argumente que  $|G_0 : V| = 2$ .
- C. Muestre cómo implementar MST-REDUCE para que se ejecute en el tiempo  $O(E)$ . (Pista: Use estructuras de datos simples.)
- d. Suponga que ejecutamos  $k$  fases de MST-REDUCE, usando la salida  $G_0$  producida por una fase como la entrada  $G$  para la siguiente fase y acumulando aristas en  $T$ . Argumente que el tiempo total de ejecución de las  $k$  fases es  $O(kE)$ .
- mi. Suponga que después de ejecutar  $k$  fases de MST-REDUCE, como en la parte (d), ejecutamos el algoritmo de Prim llamando a MST-PRIM. $G_0 ; c_0 ; r$ , donde  $G_0$  tributo  $c_0$ , con peso en , es devuelto por la última fase y  $r$  es cualquier vértice en  $G_0 : V$ . Muestre cómo seleccionar  $k$  para que el tiempo total de ejecución sea  $O(\lg \lg V)$ . Argumente que su elección de  $k$  minimiza el tiempo total de ejecución asintótica.
- F. ¿Para qué valores de  $jE_j$  (en términos de  $|V|$ ) el algoritmo de Prim con preprocesamiento supera asintóticamente al algoritmo de Prim sin preprocesamiento?

23-3 Árbol generador de cuellos de botella Un árbol generador de cuellos de botella  $T$  de un grafo no dirigido  $G$  es un árbol generador de  $G$  cuyo mayor peso de arista es mínimo sobre todos los árboles generadores de  $G$ . Decimos que el valor del árbol generador de cuellos de botella es el peso del borde de peso máximo en  $T$

- a. Argumente que un árbol de expansión mínimo es un árbol de expansión de cuello de botella.
- La parte (a) muestra que encontrar un árbol de expansión de cuello de botella no es más difícil que encontrar un árbol de expansión mínimo. En las partes restantes, mostraremos cómo encontrar un árbol de expansión de cuello de botella en tiempo lineal.
- b. Proporcione un algoritmo de tiempo lineal que, dado un gráfico  $G$  y un entero  $b$ , determine si el valor del árbol de expansión del cuello de botella es como mucho  $b$ .
- C. Use su algoritmo para la parte (b) como una subrutina en un algoritmo de tiempo lineal para el problema del árbol de expansión del cuello de botella. (Sugerencia: es posible que desee utilizar una subrutina que contrae conjuntos de aristas, como en el procedimiento MST-REDUCE descrito en el problema 23-2).

#### 23-4 Algoritmos alternativos de árbol de expansión mínima

En este problema, damos pseudocódigo para tres algoritmos diferentes. Cada uno toma un gráfico conectado y una función de peso como entrada y devuelve un conjunto de aristas T . Para cada algoritmo, demuestre que T es un árbol de expansión mínimo o demuestre que T no es un árbol de expansión mínimo. Describa también la implementación más eficiente de cada algoritmo, ya sea que calcule o no un árbol de expansión mínimo.

a. QUIZÁS-MST-AG; c/

```
1 ordenar los bordes en orden no creciente de pesos de borde w
2 TDE 3
```

```
para cada arista e, tomada en peso no creciente si T feg es un
grafo conexo 4 5
```

```
    TDT feg 6
    retorno T
```

b. QUIZÁS-MST-BG; con 1

```
TD; 2 para
```

```
    cada arista e, tomado en orden arbitrario 3 si T
```

```
        [ feg no tiene ciclos 4 TDT [ feg 5
        devuelve T
```

C. QUIZÁS-MST-CG; con 1

```
TD; 2 para
```

```
    cada arista e, tomada en orden arbitrario 3 TDT
```

```
        [ feg si T tiene un ciclo
```

```
            c 4 5 sea e0 una
```

```
            arista de peso máximo en c 6 TDT fe0 g 7 return T
```

Notas del capítulo

Tarjan [330] analiza el problema del árbol de expansión mínimo y proporciona un excelente material avanzado. Graham y Hell [151] compilaron una historia del problema del árbol de expansión mínimo.

Tarjan atribuye el primer algoritmo de árbol de expansión mínimo a un artículo de 1926 de O. Bor'uvka. El algoritmo de Borúuvka consiste en ejecutar O.lg V / iteraciones del

procedimiento MST-REDUCE descrito en el problema 23-2. El algoritmo de Kruskal fue informado por Kruskal [222] en 1956. El algoritmo comúnmente conocido como algoritmo de Prim fue inventado por Prim [285], pero también fue inventado antes por V. Jarník en 1930.

La razón subyacente por la que los algoritmos codiciosos son efectivos para encontrar árboles de expansión mínimos es que el conjunto de bosques de un gráfico forma una matroide gráfica. (Consulte la Sección 16.4.)

Cuando  $jEj D .V \lg V$ , el algoritmo de Prim, implementado con montones de Fibonacci, se ejecuta en tiempo  $O(EV)$ . Para gráficos más dispersos, usando una combinación de las ideas del algoritmo de Prim, el algoritmo de Kruskal y el algoritmo de Borúuvka, junto con estructuras de datos avanzadas, Fredman y Tarjan [114] dan un algoritmo que se ejecuta en  $O(\lg V / \text{time})$ . Gabow, Galil, Spencer y Tarjan [120] mejoraron este algoritmo para ejecutarlo en  $O(\lg \lg V / \text{time})$ . Chazelle [60] proporciona un algoritmo que se ejecuta en  $O(\lg E; V / \text{time})$ , donde  $\lg E; V$  es la inversa funcional de la función de Ackermann. (Consulte las notas del capítulo 21 para obtener una breve discusión sobre la función de Ackermann y su inversa). A diferencia de los algoritmos de árbol de expansión mínima anteriores, el algoritmo de Chazelle no sigue el método codicioso.

Un problema relacionado es la verificación del árbol de expansión, en el que se nos da un gráfico  $G; D; V; E$  y un árbol  $T$ , y deseamos determinar si  $T$  es un árbol de expansión mínimo de  $G$ . King [203] proporciona un algoritmo de tiempo lineal para verificar un árbol de expansión, basándose en el trabajo anterior de Komlós [215] y Dixon, Rauch y Tarjan [90].

Todos los algoritmos anteriores son deterministas y caen en el modelo basado en la comparación descrito en el Capítulo 8. Karger, Klein y Tarjan [195] proporcionan un algoritmo de árbol de expansión mínimo aleatorio que se ejecuta en  $O(VCE/\text{time})$ . Este algoritmo usa la recursividad de manera similar al algoritmo de selección de tiempo lineal de la Sección 9.3: una llamada recursiva a un problema auxiliar identifica un subconjunto de las aristas  $E_0$  que no pueden estar en ningún árbol de expansión mínimo. Otra llamada recursiva en  $E \setminus E_0$  luego encuentra el árbol de expansión mínimo. El algoritmo también utiliza ideas del algoritmo de Borúuvka y del algoritmo de King para la verificación del árbol de expansión.

Fredman y Willard [116] mostraron cómo encontrar un árbol de expansión mínimo en  $O(VCE/\text{time})$  utilizando un algoritmo determinista que no se basa en la comparación. Su algoritmo asume que los datos son enteros de  $b$  bits y que la memoria de la computadora consta de palabras direccionables de  $b$  bits.

## 24

## Rutas más cortas de fuente única

El profesor Patrick desea encontrar la ruta más corta posible de Phoenix a Indianápolis. Dado un mapa de carreteras de los Estados Unidos en el que está marcada la distancia entre cada par de intersecciones adyacentes, ¿cómo puede determinar esta ruta más corta?

Una forma posible sería enumerar todas las rutas de Phoenix a Indianápolis, sumar las distancias de cada ruta y seleccionar la más corta. Es fácil ver, sin embargo, que incluso descartando rutas que contengan ciclos, el profesor Patrick tendría que examinar una enorme cantidad de posibilidades, la mayoría de las cuales simplemente no vale la pena considerar. Por ejemplo, una ruta de Phoenix a Indianápolis que pasa por Seattle es obviamente una mala elección, porque Seattle está a varios cientos de millas fuera del camino.

En este capítulo y en el Capítulo 25, mostramos cómo resolver tales problemas de manera eficiente. En un problema de caminos más cortos, se nos da un gráfico dirigido y ponderado  $G = \langle V, E \rangle$ , con función de peso  $w : E \rightarrow \mathbb{R}$  asignando bordes a pesos de valor real. El peso  $w_p$  del camino  $p$  es la suma de los pesos de sus aristas constituyentes:

$$w_p = \sum_{(u, v) \in p} w(u, v)$$

Definimos el peso del camino más corto  $w_p$  / de ti a por

g si hay un camino de u a lo :  
yo.u; / D ( minfw.p / W u contrario:

El camino más corto desde el vértice  $u$  al vértice  $v$  se define entonces como cualquier camino  $p$  con peso  $w_p \leq w_p$  /  $u; v$ .

En el ejemplo de Phoenix a Indianápolis, podemos modelar el mapa de carreteras como un gráfico: los vértices representan intersecciones, los bordes representan segmentos de carretera entre intersecciones y los pesos de los bordes representan distancias de carretera. Nuestro objetivo es encontrar el camino más corto desde una intersección determinada en Phoenix hasta una intersección determinada en Indianápolis.

Los pesos de los bordes pueden representar otras métricas además de las distancias, como el tiempo, el costo, las penalizaciones, las pérdidas o cualquier otra cantidad que se acumule linealmente a lo largo de una ruta y que queramos minimizar.

El algoritmo de búsqueda primero en anchura de la Sección 22.2 es un algoritmo de caminos más cortos que funciona en gráficos no ponderados, es decir, gráficos en los que cada borde tiene un peso unitario. Debido a que muchos de los conceptos de la búsqueda primero en amplitud surgen en el estudio de las rutas más cortas en grafos ponderados, es posible que desee revisar la Sección 22.2 antes de continuar.

#### variantes

En este capítulo, nos centraremos en el problema de los caminos más cortos de fuente única: dado un gráfico  $G = \langle V, E \rangle$ , queremos encontrar el camino más corto desde un vértice fuente dado  $s \in V$  a cada vértice  $t \in V$ . El algoritmo para el problema de fuente única puede resolver muchos otros problemas, incluidas las siguientes variantes.

**Problema de rutas más cortas de destino único:** Encuentre una ruta más corta a un vértice de destino dado  $t$  desde cada vértice. Al invertir la dirección de cada borde en el gráfico, podemos reducir este problema a un problema de fuente única.

**Problema de la ruta más corta de un solo par:** Encuentre la ruta más corta desde  $u$  hasta para los vértices dados  $u$  y  $v$ . Si resolvemos el problema de fuente única con el vértice fuente  $u$ , también resolvemos este problema. Además, todos los algoritmos conocidos para este problema tienen el mismo tiempo de ejecución asintótico en el peor de los casos que los mejores algoritmos de fuente única.

**Problema de caminos más cortos de todos los pares:** encuentre el camino más corto desde  $u$  hasta para cada par de vértices  $u$  y  $v$ . Aunque podemos resolver este problema ejecutando un algoritmo de fuente única una vez desde cada vértice, generalmente podemos resolverlo más rápido. Además, su estructura es interesante por derecho propio. El capítulo 25 aborda el problema de todos los pares en detalle.

#### Subestructura óptima de un camino más corto

Los algoritmos de rutas más cortas generalmente se basan en la propiedad de que una ruta más corta entre dos vértices contiene otras rutas más cortas dentro de ella. (El algoritmo de flujo máximo de Edmonds-Karp del capítulo 26 también se basa en esta propiedad). Recuerde que la subestructura óptima es uno de los indicadores clave de que la programación dinámica (capítulo 15) y el método codicioso (capítulo 16) podrían aplicarse. El algoritmo de Dijkstra, que veremos en la Sección 24.3, es un algoritmo codicioso, y el algoritmo de Floyd Warshall, que encuentra los caminos más cortos entre todos los pares de vértices (consulte la Sección 25.2), es un algoritmo de programación dinámica. El siguiente lema establece la propiedad de subestructura óptima de los caminos más cortos con mayor precisión.

Lema 24.1 (Los subcaminos de los caminos más cortos son los caminos más cortos)

Dado un grafo dirigido y ponderado GD .V; E/ con función de peso w WE ! R, sea p D h0; 1;:::;ki sea el camino más corto de vértice k y, para cualquier i y j tal que 0 ijk, sea pij D hi; iC1;:::;j i el subcamino de p desde el vértice i hasta el vértice j . Entonces, pij es el camino más corto de i a j .

Prueba Si descomponemos el camino p  $\begin{matrix} 0 & \xrightarrow{\text{pag}} & 0i & \xrightarrow{\text{pij}} & i & \xrightarrow{\text{pij}} & j & \xrightarrow{\text{pjk}} & k \end{matrix}$ , entonces tenemos eso en wp/ D w.p0i/ C w.pij / C w.pjk/. Ahora, suponga que hay un camino p0  $\begin{matrix} 0 & \xrightarrow{\text{de ij}} & i & \xrightarrow{\text{p0}} & j & \xrightarrow{\text{pjk}} & k \end{matrix}$  de ij a k cuyo peso es w.p0i/w.p0j/w.p0k/. Entonces, 0 a j ij a  $\begin{matrix} 0 & \xrightarrow{\text{pag}} & 0i & \xrightarrow{\text{p0}} & j & \xrightarrow{\text{pjk}} & k \end{matrix}$  es un camino de 0  $y_0/Cw.pjk/$  es menor que wp/, lo que contradice la suposición de que p es el camino más corto desde 0 a k. ■

### Bordes de peso negativo

Algunas instancias del problema de los caminos más cortos de fuente única pueden incluir bordes cuyos pesos son negativos. Si el gráfico GD .V; E/ no contiene ciclos de peso negativos accesibles desde la fuente s, entonces para todos los 2 V el peso de camino más corto i.s; / queda bien definido, incluso si tiene un valor negativo. Sin embargo, si el gráfico contiene un ciclo de ponderación negativa accesible desde s, las ponderaciones del camino más corto no están bien definidas. Ningún camino desde s hasta un vértice en el ciclo puede ser un camino más corto; siempre podemos encontrar un camino con menor peso siguiendo el camino "más corto" propuesto y luego recorriendo el ciclo de peso negativo. Si hay un ciclo de peso negativo en algún camino desde s hasta definimos i.s; / D 1.

La figura 24.1 ilustra el efecto de las ponderaciones negativas y los ciclos de ponderaciones negativas sobre las ponderaciones del camino más corto. Como solo hay un camino de s a a (el camino hs; ai), tenemos i.s; a/ D ws; a/ D 3. Del mismo modo, sólo hay un camino de s a b, y por lo tanto i.s; b/ D ws; a/ C agua; b/ D 3 C .4/ D 1. Hay infinitos caminos de s a c: hs; ci, hs; C; d; ci, hs; C; d; C; d; ci, y así sucesivamente.

Como el ciclo hc;d;ci tiene peso 6 C .3/D 3>0, el camino más corto de s a c es hs; ci, con peso i.s; c/ D ws; c/ D 5. De manera similar, el camino más corto de s a d es hs;c;di, con peso i.s; d/ D ws; c/Cw.c; d/ D 11. Análogamente, hay infinitos caminos de s a e: hs; ei, hs; mi; F; ei, hs; mi; F; mi; F; ei, y así sucesivamente. Porque el ciclo él; F; ei tiene peso 3 C .6/ D 3<0, sin embargo, no hay un camino más corto de s a e. Atravesando el ciclo de peso negativo él; F; ei arbitrariamente muchas veces, podemos encontrar caminos de s a e con pesos negativos arbitrariamente grandes, y así i.s; e/ D 1. Del mismo modo, i.s; f / D 1. Debido a que g es alcanzable desde f , también podemos encontrar caminos con pesos negativos arbitrariamente grandes de s a g, y así i.s; g/ D 1. Los vértices h, ij tambien forman un ciclo de peso negativo. Sin embargo, no son accesibles desde s, por lo que i.s; h/ D i.s; i / D i.s; j / D 1.

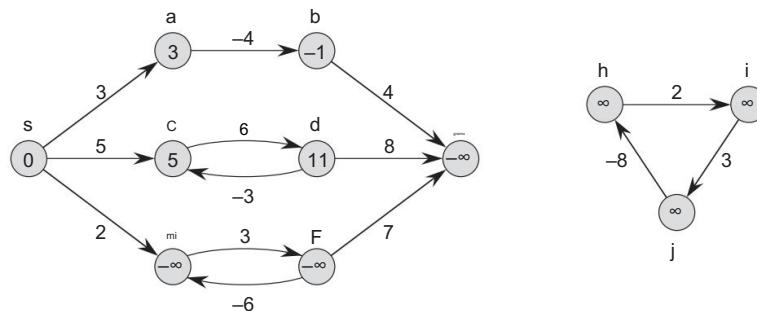


Figura 24.1 Pesos de los bordes negativos en un gráfico dirigido. El peso de la ruta más corta desde la fuente s aparece dentro de cada vértice. Debido a que los vértices e y f forman un ciclo de peso negativo al que se puede llegar desde s, tienen un peso de camino más corto de 1. Como el vértice g es accesible desde un vértice cuyo peso de camino más corto es 1, también tiene un peso de camino más corto de 1. Los vértices como h, i y j no son accesibles desde s, por lo que sus pesos de ruta más corta son 1, aunque se encuentran en un ciclo de peso negativo.

Algunos algoritmos de rutas más cortas, como el algoritmo de Dijkstra, asumen que todos los pesos de borde en el gráfico de entrada no son negativos, como en el ejemplo del mapa de ruta. Otros, como el algoritmo de Bellman-Ford, permiten bordes de ponderación negativa en el gráfico de entrada y producen una respuesta correcta siempre que no haya ciclos de ponderación negativa accesibles desde la fuente. Por lo general, si existe un ciclo de peso negativo de este tipo, el algoritmo puede detectar e informar de su existencia.

### Ciclos

¿Puede un camino más corto contener un ciclo? Como acabamos de ver, no puede contener un ciclo de peso negativo. Tampoco puede contener un ciclo de peso positivo, ya que al eliminar el ciclo del camino se produce un camino con los mismos vértices de origen y destino y un peso de camino más bajo. Es decir, si  $p \in h_0; 1; \dots; k_i$  es un camino y  $c \in h_i; iC1; \dots; j$  es un ciclo de peso positivo en este camino (de modo que  $w_c > 0$ ), entonces el camino  $p0 \in h_0; 1; \dots; i; j@1; jC2; \dots; k_i$  tiene un peso  $w_p / D w_p / w_c < w_p /$ , por lo que  $p$  no puede ser el camino más corto de  $k$ .

Eso deja solo ciclos de peso 0. Podemos eliminar un ciclo de peso 0 de cualquier camino para producir otro camino cuyo peso sea el mismo. Por lo tanto, si hay un camino más corto desde un vértice de origen  $s$  hasta un vértice de destino que contiene un ciclo de peso 0, entonces hay otro camino más corto desde  $s$  hasta sin este ciclo. Siempre que una ruta más corta tenga ciclos de peso 0, podemos eliminar repetidamente estos ciclos de la ruta hasta que tengamos una ruta más corta que no tenga ciclos. Por lo tanto, sin pérdida de generalidad podemos suponer que cuando buscamos caminos más cortos, estos no tienen ciclos, es decir, son caminos simples. Dado que cualquier camino acíclico en un gráfico  $G = \langle V, E \rangle$

contiene como máximo  $jV$  vértices distintos, también contiene como máximo  $jV$  1 aristas. Por lo tanto, podemos restringir nuestra atención a los caminos más cortos de como mucho  $jV$  1 aristas.

#### Representación de los caminos más cortos

A menudo deseamos calcular no solo los pesos de los caminos más cortos, sino también los vértices de los caminos más cortos. Representamos los caminos más cortos de manera similar a como representamos los árboles de ancho primero en la Sección 22.2. Dado un grafo  $G = \langle V, E \rangle$ , mantenemos para cada vértice  $s \in V$  un  $f_s$  que es otro vértice o NIL. El predecesor : los algoritmos de caminos más cortos de este capítulo establecen los atributos de modo que la cadena de predecesores que se origina en un vértice corre hacia atrás a lo largo de procedimiento  $\text{PRINT-PATH}(G, s)$ . Así, un camino más corto desde  $s$  hasta  $v$  :  $\pi$  NIL, el dado un vértice para el cual de la Sección 22.2 imprimiremos un camino más corto de  $s$  a  $v$ .

Sin embargo, en medio de la ejecución de un algoritmo de rutas más cortas, es posible que los valores no indiquen las rutas más cortas. Como en la búsqueda primero en anchura, nos interesarán el subgrafo predecesor  $G_f = \langle V_f, E_f \rangle$  inducida por los valores. Aquí nuevamente, definimos el conjunto de vértices  $V_f$  como el conjunto de vértices de  $G$  con predecesores que no son NIL, más la fuente  $s$ :

$V_f = \{v \in V \mid \exists u \in V \text{ tal que } f_u = v\}$

El conjunto de aristas dirigidas  $E_f$  es el conjunto de aristas inducidas por los valores de los vértices en  $V_f$ :

$E_f = \{(u, v) \in E \mid u \in V_f \text{ y } v \in V_f \text{ y } f_u = v\}$

Demostraremos que los valores producidos por los algoritmos de este capítulo tienen la propiedad de que en la terminación  $G$  hay un “árbol de caminos más cortos”, informalmente, un árbol con raíces que contiene un camino más corto desde la fuente  $s$  hasta cada vértice que es accesible desde  $s$ . Un árbol de caminos más cortos es como el árbol de ancho primero de la Sección 22.2, pero contiene los caminos más cortos desde la fuente definida en términos de pesos de borde en lugar de números de bordes. Para ser precisos, sea  $G = \langle V, E \rangle$  sea un gráfico dirigido y ponderado con una función de peso  $w : E \rightarrow \mathbb{R}$ , y suponga que  $G$  no contiene ciclos de peso negativo accesibles desde el vértice fuente  $s \in V$ , de modo que los caminos más cortos estén bien definidos. Un árbol de caminos más cortos con raíz en  $s$  es un subgrafo dirigido  $G_0 = \langle V_0, E_0 \rangle$ , donde  $V_0 \subseteq V$  y  $E_0 \subseteq E$ , tal que

1.  $V_0$  es el conjunto de vértices alcanzable desde  $s$  en  $G$ ,
2.  $G_0$  forma un árbol enraizado con raíz  $s$ , y
3. para todos los  $v \in V_0$ , el único camino simple desde  $s$  hasta  $v$  en  $G_0$  es el camino más corto desde  $s$  a  $v$  en  $G$ .

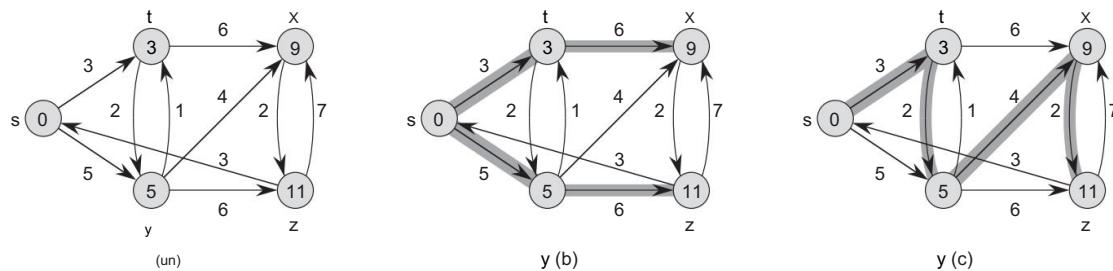


Figura 24.2 (a) Un gráfico dirigido ponderado con pesos de ruta más corta desde la fuente s. (b) Los bordes sombreados forman un árbol de caminos más cortos con raíz en la fuente s. (c) Otro árbol de caminos más cortos con la misma raíz.

Los caminos más cortos no son necesariamente únicos, y tampoco lo son los árboles de caminos más cortos. Por ejemplo, la figura 24.2 muestra un gráfico dirigido ponderado y dos árboles de caminos más cortos con la misma raíz.

### Relajación

Los algoritmos de este capítulo utilizan la técnica de la relajación. Para cada vértice  $v$ , mantenemos un atributo  $:d$ , que es un límite superior en el peso de un camino más corto desde la fuente s hasta  $v$ . Llamamos a  $:d$  una estimación del camino más corto. Inicializamos las estimaciones del camino más corto y los predecesores mediante el siguiente procedimiento ,V /-time:

```
INICIALIZAR-UNA-FUENTE.G; s/ 1 por
cada vértice 2 G:V
2          :d D 1
           D NULO
3 4 s:d D 0
```

Después de la inicialización, D NIL para todos los 2 V, s:d D 0, y :d D 1 para tenemos 2 V fsg.

El proceso de relajación de un borde  $u$ ; / consiste en probar si podemos mejorar el camino más corto hasta el momento pasando por  $u$  y, si es así, actualizar Un paso de relajación 1 puede disminuir el valor del camino más corto ing :d y

<sup>1</sup>Puede parecer extraño que el término "relajación" se utilice para una operación que aprieta un límite superior. El uso del término es histórico. El resultado de un paso de relajación puede verse como una relajación de la restricción  $:d \leq C_{wu} /$ , que, por la desigualdad del triángulo (Lema 24.10), debe satisfacerse si  $u:d \leq D_{i.s}; u /:d \leq D_{i.s} /$ . Es decir, si  $:d \leq C_{wu} /$ , no hay "presión" para satisfacer esta restricción, por lo que la restricción es "relajada".

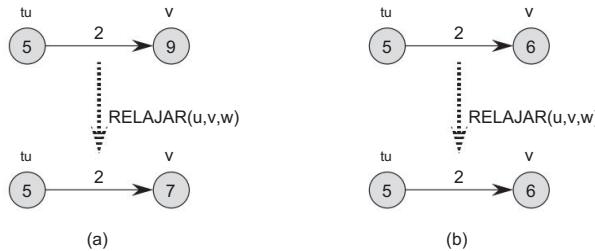


Figura 24.3 Relajación de un borde  $.u;$  / con peso  $wu;$  /  $D 2.$  La estimación del camino más corto de cada vértice aparece dentro del vértice. (a) Porque  $:d > u:d C wu;$  / antes de la relajación, el valor de  $:d$  disminuye. (b) Aquí,  $:d = u:d C wu;$  / antes de relajar el borde, por lo que el paso de relajación deja  $:d$  sin cambios.

estimar  $:d$  y actualizar el atributo predecesor forma

El siguiente código por

un paso de relajación en el borde  $.u;$  / en  $O.1/t$  tiempo:

```
RELAX.u; ; w/ 1
si :d > u:d C wu; / :d D u:d C
2      wu; /
3      tu
```

La figura 24.3 muestra dos ejemplos de relajación de un borde, uno en el que disminuye la estimación de la ruta más corta y otro en el que no cambia ninguna estimación.

Cada algoritmo en este capítulo llama INITIALIZE-SINGLE-SOURCE y luego relaja los bordes repetidamente. Además, la relajación es el único medio por el cual cambian las estimaciones del camino más corto y los predecesores. Los algoritmos de este capítulo difieren en la cantidad de veces que relajan cada borde y el orden en que relajan los bordes. El algoritmo de Dijkstra y el algoritmo de caminos más cortos para gráficos acíclicos dirigidos relajan cada borde exactamente una vez. El algoritmo de Bellman-Ford relaja cada arista  $jV$   $j 1$  veces.

Propiedades de los caminos más cortos y relajación.

Para probar que los algoritmos de este capítulo son correctos, recurriremos a varias propiedades de los caminos más cortos y la relajación. Enunciamos estas propiedades aquí, y la Sección 24.5 las prueba formalmente. Para su referencia, cada propiedad indicada aquí incluye el lema o número de corolario apropiado de la Sección 24.5. Las últimas cinco de estas propiedades, que se refieren a las estimaciones de la ruta más corta o al subgráfo predecesor, suponen implícitamente que el gráfico se inicializa con una llamada a INITIALIZE SINGLE-SOURCE.G; s/ y que la única forma en que las estimaciones del camino más corto y el subgrafo predecesor cambian es mediante alguna secuencia de pasos de relajación.

## Desigualdad triangular (Lema 24.10)

Para cualquier arista  $s; t \in E$ , tenemos  $d(s; t) \leq d(s; u) + d(u; t)$ .

## Propiedad de límite superior (Lema 24.11)

Siempre tenemos  $d(s; t) \leq 2|V|$  para todos los vértices  $s, t \in V$ , y una vez que logra el límite superior, nunca cambia.

## Propiedad sin camino (Corolario 24.12)

Si no hay camino de  $s$  a  $t$ , entonces siempre tenemos  $d(s; t) = \infty$ .

Propiedad de convergencia (Lema 24.14) y si  $s; t \in V$ 

$d(s; t)$  para alguna  $s, t \in V$  en cualquier momento ~~desde relajación~~ es un camino más corto en el borde  $s; t$ , luego  $d(s; t)$  en todo momento después.

## Propiedad de relajación del camino (Lema 24.15)

Si  $p$  es el camino más corto desde  $s$  a  $t$  y relajamos las aristas de  $p$  hasta  $k$  en el paso  $i$ , entonces  $d(s; t) \leq d(s; k) + d(k; t)$ .

Esta propiedad se mantiene independientemente de cualquier otro paso de relajación que ocurra, incluso si están entremezclados con relajaciones de los bordes de  $p$ .

## Propiedad de antecesores-subgrafo (Lema 24.17)

Una vez que  $d(s; t) = \infty$  para todos  $s, t \in V$ , el subgrafo predecesor es un camino más corto árbol enraizado en  $s$ .

## Bosquejo del capítulo

La sección 24.1 presenta el algoritmo de Bellman-Ford, que resuelve el problema de los caminos más cortos de fuente única en el caso general en el que los bordes pueden tener un peso negativo. El algoritmo de Bellman-Ford es notablemente simple y tiene el beneficio adicional de detectar si se puede alcanzar un ciclo de peso negativo desde la fuente. La sección 24.2 proporciona un algoritmo de tiempo lineal para calcular los caminos más cortos desde una sola fuente en un gráfico acíclico dirigido. La sección 24.3 cubre el algoritmo de Dijkstra, que tiene un tiempo de ejecución menor que el algoritmo de Bellman-Ford pero requiere que los pesos de los bordes no sean negativos. La sección 24.4 muestra cómo podemos usar el algoritmo de Bellman-Ford para resolver un caso especial de programación lineal. Finalmente, la Sección 24.5 prueba las propiedades de los caminos más cortos y la relajación mencionadas anteriormente.

Requerimos algunas convenciones para hacer aritmética con infinitos. Supondremos que para cualquier número real  $a \neq 1$ , tenemos un  $C_1 D_1 C$  a  $D_1$ . Además, para que nuestras demostraciones sean válidas en presencia de ciclos de peso negativo, supondremos que para cualquier número real  $a \neq 1$ , tener un  $C_1 D_1 C$  a  $D_1$ .

Todos los algoritmos de este capítulo suponen que el gráfico dirigido  $G$  se almacena en la representación de lista de adyacencia. Además, se almacena con cada borde su peso, de modo que a medida que recorremos cada lista de adyacencia, podemos determinar los pesos de los bordes en O(1/tiempo por borde).

## 24.1 El algoritmo Bellman-Ford

El algoritmo de Bellman-Ford resuelve el problema de los caminos más cortos de fuente única en el caso general en el que los pesos de los bordes pueden ser negativos. Dado un gráfico dirigido y ponderado  $G = \langle V, E \rangle$  con fuente  $s$  y función ponderada  $w : E \rightarrow \mathbb{R}$ , el algoritmo Bellman-Ford devuelve un valor booleano que indica si hay o no un ciclo de peso negativo al que se puede acceder desde la fuente. Si existe tal ciclo, el algoritmo indica que no existe solución. Si no existe tal ciclo, el algoritmo produce los caminos más cortos y sus pesos.

El algoritmo relaja los bordes, disminuyendo progresivamente una estimación  $d$  en el peso de la ruta más corta desde la fuente  $s$  hasta cada vértice  $v \in V$  hasta que alcanza el peso real de la ruta más corta  $s \rightarrow v$ . El algoritmo devuelve VERDADERO si y solo si el gráfico no contiene ciclos de peso negativo a los que se pueda acceder desde la fuente.

```
BELLMAN-FORD(G; w; s) {
    INICIALIZAR-UNA-FUENTE(G; s);
    for i = 1 to |V| - 1 do
        for each edge u → v in G do
            RELAX(u; v; w);
    if there is a negative-weight cycle then
        return FALSE;
    else
        return TRUE;
}
```

La figura 24.4 muestra la ejecución del algoritmo de Bellman-Ford en un gráfico con 5 vértices. Después de inicializar los valores de  $d$  de todos los vértices en la línea 1, el algoritmo hace que  $j$  pase por los bordes del gráfico. Cada paso es una iteración del ciclo for de las líneas 2 a 4 y consiste en relajar cada borde del gráfico una vez. Las figuras 24.4(b)–(e) muestran el estado del algoritmo después de que cada uno de los cuatro pase por los bordes. Después de hacer  $j$  pasos, las líneas 5 a 8 verifican si hay un ciclo de peso negativo y devuelven el valor booleano apropiado. (Veremos un poco más adelante por qué funciona esta verificación).

El algoritmo de Bellman-Ford se ejecuta en el tiempo  $O(|V||E|)$ , ya que la inicialización en la línea 1 toma  $|V|$  tiempo, cada uno de los  $|V|$  pasos por los bordes en las líneas 2 a 4 toma  $|E|$  tiempo, y el para el bucle de las líneas 5–7 se necesita tiempo  $O(E)$ .

Para probar la corrección del algoritmo de Bellman-Ford, comenzamos mostrando que si no hay ciclos de peso negativo, el algoritmo calcula los pesos correctos del camino más corto para todos los vértices accesibles desde la fuente.

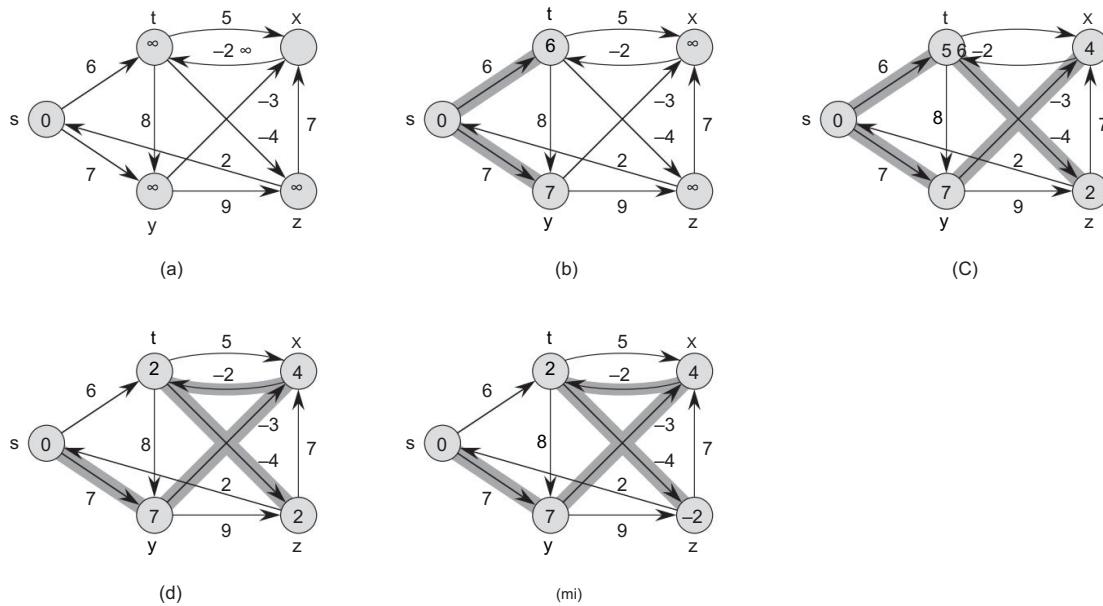


Figura 24.4 La ejecución del algoritmo Bellman-Ford. La fuente es el vértice  $s$ . Los valores  $d$  aparecen dentro de los vértices, y los bordes sombreados indican valores predecesores: if edge  $u; v$  está sombreado, luego  $D[u] = v$ . En este ejemplo particular, cada pasada relaja los bordes en el orden  $t; x; y; t; z; x; t; y; x; y; z; s; t; s; y; t; z; x; y; z$ . (a) La situación justo antes de la primera pasada por los bordes. (b)–(e) La situación después de cada pasada sucesiva sobre los bordes. Los valores  $d$  y en la parte (e) son los valores finales. El algoritmo Bellman-Ford devuelve VERDADERO en este ejemplo.

### Lema 24.2

Sea  $G = \langle V, E \rangle$  un gráfico dirigido y ponderado con fuente  $s$  y función de ponderación  $w : E \rightarrow \mathbb{R}$ , y suponga que  $G$  no contiene ciclos de peso negativo que sean accesibles desde  $s$ . Luego, después de las iteraciones  $jV \leq 1$  del bucle for de las líneas 2 a 4 de BELLMAN-FORD, tenemos  $\forall v \in V : d[v] = \text{distancia mínima de } s \text{ a } v$ .

**Demuestra** Probamos el lema apelando a la propiedad de relajación del camino.

Considere cualquier vértice que sea alcanzable desde  $s$ , y sea  $p = s; v_1; v_2; \dots; v_k$ , donde sea  $v_0 = s$  y  $v_k = v$ , el camino más corto de  $s$  a  $v$ . Como los caminos más cortos son simples,  $p$  tiene como máximo  $jV \leq 1$  aristas, por lo que  $k \leq jV \leq 1$ . Cada una de las  $jV \leq 1$  iteraciones del ciclo for de las líneas 2–4 relaja todas las  $e \in E$  aristas. Entre las aristas relajadas en la  $i$ -ésima iteración, para  $i \in \{1, 2, \dots, k\}$ , es  $e_i = (v_{i-1}, v_i)$ . Por la propiedad de relajación de trayectoria, por lo tanto,  $d[v] \leq d[v_{i-1}] + w(e_i)$ . ■

## Corolario 24.3

Sea  $G = (V, E)$  un grafo dirigido ponderado con vértice fuente  $s$  y función de ponderación  $w : V \times V \rightarrow \mathbb{R}$ , y suponga que  $G$  no contiene ciclos de peso negativo que sean accesibles desde  $s$ . Entonces, para cada vértice  $v \in V$  hay un camino de  $s$  a  $v$  y sólo si BELLMAN-FORD termina en  $d_v < \infty$  cuando se ejecuta en  $G$ .

Prueba La prueba se deja como Ejercicio 24.1-2. ■

## Teorema 24.4 (Corrección del algoritmo Bellman-Ford)

Sea BELLMAN-FORD sobre un grafo dirigido ponderado  $G = (V, E)$  con fuente  $s$  y función ponderada  $w : V \times V \rightarrow \mathbb{R}$ . Si  $G$  no contiene ciclos de peso negativo que sean accesibles desde  $s$ , entonces el algoritmo devuelve VERDADERO, tenemos  $d_v < \infty$  para todos los vértices  $v \in V$  y el subgrafo predecesor  $G$  es un árbol de caminos más cortos con raíz en  $s$ . Si  $G$  contiene un ciclo de peso negativo accesible desde  $s$ , entonces el algoritmo devuelve FALSO.

Prueba Suponga que el gráfico  $G$  no contiene ciclos de peso negativo que sean accesibles desde la fuente  $s$ . Primero probamos la afirmación de que en la terminación,  $d_v < \infty$  para todos los vértices  $v \in V$ . Si se puede alcanzar el vértice  $v$  desde  $s$ , entonces el Lema 24.2 prueba esta afirmación. Si no es accesible desde  $s$ , entonces el reclamo se deriva de la propiedad sin ruta. Por lo tanto, la afirmación queda probada. La propiedad del subgrafo predecesor, junto con la afirmación, implica que  $G$  es un árbol de caminos más cortos. Ahora usamos la afirmación para mostrar que BELLMAN-FORD devuelve VERDADERO. En la terminación, tenemos para todos los bordes  $e = (u, v) \in E$ ,

$$d_u < \infty \text{ y } d_v < \infty$$

(por la desigualdad triangular)

$$d_u \leq d_v + w(u, v)$$

y así ninguna de las pruebas en la línea 6 hace que BELLMAN-FORD devuelva FALSO. Allá por lo tanto, devuelve VERDADERO.

Ahora, suponga que el gráfico  $G$  contiene un ciclo de peso negativo al que se puede acceder desde la fuente  $s$ ; sea este ciclo  $c = (v_0, v_1, \dots, v_k)$ , donde  $v_0 = v_k$ . Entonces,

$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}) < 0 \quad (24.1)$$

Asumir con el propósito de contradicción que el algoritmo de Bellman-Ford devuelve  $d_{v_k} < \infty$ . Lo tanto,  $d_{v_0} < \infty$  y para  $i = 1, 2, \dots, k$ . Resumiendo lo VERDADERO. Por desigualdades alrededor del ciclo  $c$  nos da

$$\begin{array}{ccc}
 & k & \\
 X & \text{yo: d} & X \\
 iD1 & & iD1 \\
 & k & k \\
 \text{DX} & i1:d & \text{CX} w.i1; i/ \\
 iD1 & & iD1
 \end{array}$$

Desde  $\text{O}^{\text{D}}$   $k$ , cada vértice en  $C$  aparece exactamente una vez en cada una de las sumatorias  $iD_1$  y  $Pk_1:d$ , y así

$$\begin{array}{ccc}
 & k & \\
 X & i: d & \text{DX} \\
 iD1 & & iD1 \\
 & i1:d & :
 \end{array}$$

Además, por el Corolario 24.3,  $i:d$  es finito para  $i \in D_1; 2; \dots; k$ . Así,

$$\begin{array}{ccc}
 & k & \\
 0 & X & w.i1; i/ ; \\
 & iD1 &
 \end{array}$$

lo que contradice la desigualdad (24.1). Concluimos que el algoritmo de Bellman-Ford devuelve VERDADERO si el gráfico  $G$  no contiene ciclos de peso negativo accesibles desde la fuente, y FALSO en caso contrario. ■

### Ejercicios

#### 24.1-1

Ejecute el algoritmo de Bellman-Ford en la gráfica dirigida de la figura 24.4, usando el vértice  $'$  como fuente. En cada pasada, relaje los bordes en el mismo orden que en la figura y muestre los valores de  $d$  y después de cada pasada. Ahora, cambia el peso de la arista  $'x/a$  a 4 y vuelve a ejecutar el algoritmo, utilizando  $s$  como fuente.

#### 24.1-2

Demostrar Corolario 24.3.

#### 24.1-3

Dado un grafo dirigido ponderado  $GD(V, E)$  sin ciclos de peso negativo, sea  $m$  el máximo sobre todos los vértices  $v \in V$  del número mínimo de aristas en el camino más corto desde la fuente  $s$  hasta  $v$ . (Aquí, el camino más corto es por peso, no por el número de aristas). Sugiera un cambio simple al algoritmo de Bellman-Ford que le permita terminar en  $m+1$  pasos, incluso si  $m$  no se conoce de antemano.

#### 24.1-4

Modifique el algoritmo de Bellman-Ford para que establezca  $:d$  en 1 para todos los vértices para los que hay un ciclo de peso negativo en algún camino desde la fuente hasta  $v$ .

## 24.1-5 ?

Sea  $G = (V, E)$  un gráfico dirigido y ponderado con una función de peso  $w : E \rightarrow \mathbb{R}$

Dar un algoritmo O( $|V|$ ) para encontrar, para cada vértice  $v \in V$  el valor  $f(v) = \min_{u \in V} w(u, v)$ .

## 24.1-6 ?

Suponga que un grafo dirigido y ponderado  $G = (V, E)$  tiene un ciclo de peso negativo.

Proporcione un algoritmo eficiente para enumerar los vértices de uno de esos ciclos. Demuestra que tu algoritmo es correcto.

## 24.2 Caminos más cortos de fuente única en gráficos acíclicos dirigidos

Relajando los bordes de un dag ponderado (gráfico acíclico dirigido)  $G = (V, E)$  de acuerdo con una ordenación topológica de sus vértices, podemos calcular los caminos más cortos desde una sola fuente en  $O(|V|)$  tiempo. Las rutas más cortas siempre están bien definidas en un dag, ya que incluso si hay bordes de peso negativo, no pueden existir ciclos de peso negativo.

El algoritmo comienza clasificando topológicamente el dag (consulte la Sección 22.4) para imponer un ordenamiento lineal en los vértices. Si el dag contiene una ruta desde el vértice  $u$  hasta vértice  $v$ , entonces  $u$  precede en el ordenamiento topológico. Hacemos solo una pasada sobre los vértices en el orden ordenado topológicamente. A medida que procesamos cada vértice, relajamos cada borde que sale del vértice.

DAG-RUTAS MÁS CORTAS. $G; w; s/ 1$

ordenar topológicamente los vértices de  $G$  2

INITIALIZE-SINGLE-SOURCE. $G; s/ 3$  para cada

vértice  $u$ , tomado en orden ordenado topológico 4 para cada

vértice  $v \in G$ :  $Adj[u] \setminus \{u\}$  5 RELAX. $u; v; c/$

La figura 24.5 muestra la ejecución de este algoritmo.

El tiempo de ejecución de este algoritmo es fácil de analizar. Como se muestra en la Sección 22.4, la ordenación topológica de la línea 1 toma  $O(|V|)$  tiempo. La llamada de INITIALIZE SINGLE-SOURCE en la línea 2 toma  $O(|V|)$  tiempo. El bucle for de las líneas 3 a 5 hace una iteración por vértice. En total, el bucle for de las líneas 4 y 5 relaja cada arista exactamente una vez. (Aquí hemos utilizado un análisis agregado). Debido a que cada iteración del ciclo for interno toma  $O(1)$  tiempo, el tiempo de ejecución total es  $O(|V|)$ , que es lineal en el tamaño de una representación de lista de adyacencia del gráfico.

El siguiente teorema muestra que el procedimiento DAG-SHORTEST-PATHS calcula correctamente los caminos más cortos.

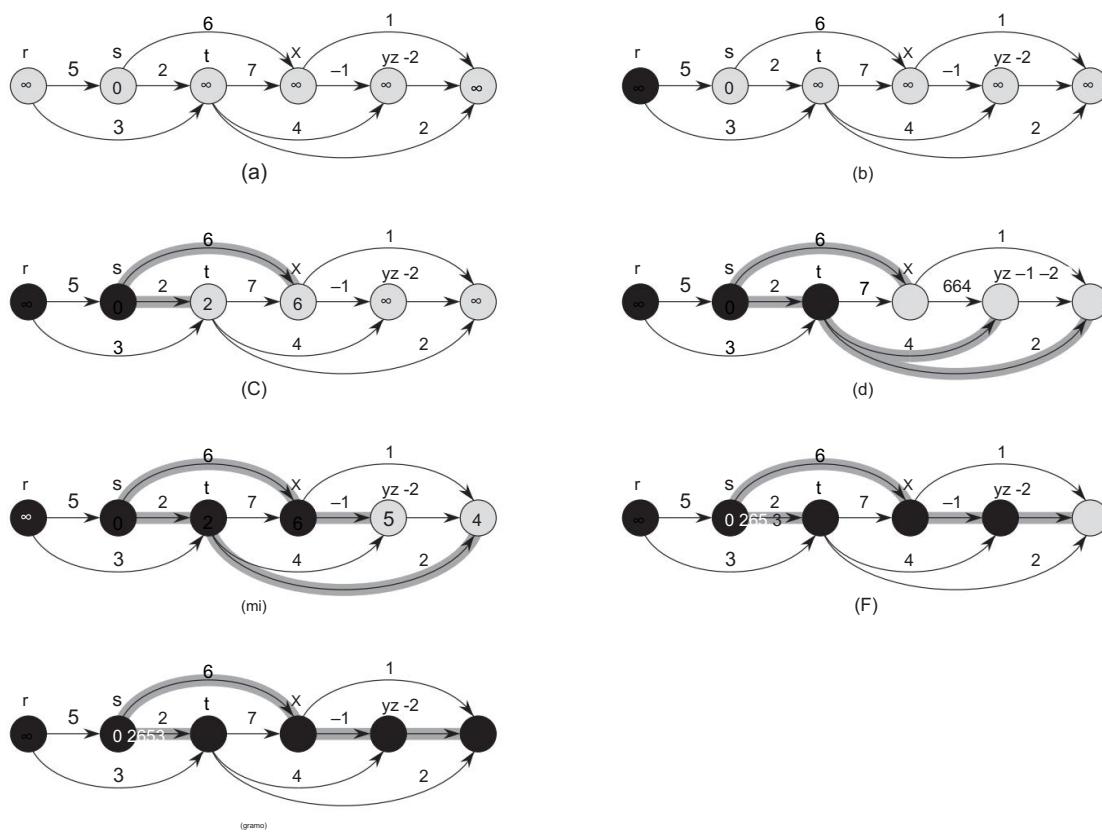


Figura 24.5 La ejecución del algoritmo para caminos más cortos en un gráfico acíclico dirigido. Los vértices están ordenados topológicamente de izquierda a derecha. El vértice fuente es s. Los valores d aparecen dentro de los vértices y los bordes sombreados indican los valores. (a) La situación antes de la primera iteración del ciclo for de las líneas 3–5. (b)–(g) La situación después de cada iteración del ciclo for de las líneas 3–5. El vértice recién ennegrecido en cada iteración se usó como u en esa iteración. Los valores mostrados en la parte (g) son los valores finales.

### Teorema 24.5

Si un grafo dirigido ponderado  $G = \langle V, E \rangle$  tiene vértices fuente s y no tiene ciclos, luego al final del procedimiento DAG-SHORTEST-PATHS,  $d[u]$  para todos los vértices  $v \in V$ , y el subgrafo predecesor G es un árbol de caminos más cortos.

Prueba Primero demostramos que  $d[u] = \text{shortest path from } s \text{ to } u$  para todos los vértices  $v \in V$  en terminación. Si no es accesible desde s, entonces  $d[u] = \infty$  por la propiedad de no-camino. Ahora, supongamos que es accesible desde s, de modo que existe un camino estrecho  $s - v_0 - v_1 - \dots - v_k$ , donde  $v_0 = s$ . Porque nosotros pro-

cesa los vértices en orden ordenado topológicamente, relajamos las aristas en  $p$  en el orden  $.0; 1/; .1;$   
 $2/; : : ; .k1; k/$ . La propiedad de relajación del camino implica que  $i:d D i.s; i/$  en la terminación para  $i D$   
 $0; 1; : : ; k$ . Finalmente, por la propiedad del subgrafo predecesor,  $G$  es un árbol de caminos más  
cortos. ■

Una aplicación interesante de este algoritmo surge en la determinación de rutas críticas en el análisis PERT chart<sup>2</sup>. Los bordes representan los trabajos a realizar y los pesos de los bordes representan los tiempos requeridos para realizar trabajos particulares. Si borde  $.u; /$  entra vértice y arista  $; x/$  deja el trabajo  $.u; /$  debe realizarse antes del trabajo  $; X/$ .

Una ruta a través de este dag representa una secuencia de trabajos que deben realizarse en un orden particular. Una ruta crítica es una ruta más larga a través del dag, correspondiente al tiempo más largo para realizar cualquier secuencia de trabajos. Por lo tanto, el peso de una ruta crítica proporciona un límite inferior del tiempo total para realizar todos los trabajos. Podemos encontrar una ruta crítica ya sea por

negando los pesos de borde y ejecutando DAG-SHORTEST-PATHS, o ejecutando

DAG-SHORTEST-PATHS, con la modificación de que reemplazamos " $1$ " por " $1$ " en la línea 2 de INITIALIZE-SINGLE-SOURCE y " $>$ " por " $<$ " en el procedimiento RELAX.

## Ejercicios

### 24.2-1

Ejecute DAG-SHORTEST-PATHS en la gráfica dirigida de la figura 24.5, usando el vértice  $r$  como fuente.

### 24.2-2

Supongamos que cambiamos la línea 3 de DAG-SHORTEST-PATHS para leer

3 para el primer  $j \in j$       1 vértices, tomados en orden clasificado topológicamente

Demuestre que el procedimiento seguiría siendo correcto.

### 24.2-3

La formulación de la gráfica PERT dada anteriormente es algo antinatural. En una estructura más natural, los vértices representarían trabajos y los bordes representarían restricciones de secuencia; es decir, borde  $.u; /$  indicaría que el trabajo  $u$  debe realizarse antes que el trabajo. Entonces asignaríamos pesos a los vértices, no a los bordes. Modifique el procedimiento DAG SHORTEST-PATHS para que encuentre la ruta más larga en un gráfico acíclico dirigido con vértices ponderados en tiempo lineal.

---

<sup>2</sup> "PERT" es un acrónimo de "técnica de revisión y evaluación de programas".

## 24.2-4

Proporcione un algoritmo eficiente para contar el número total de caminos en un gráfico acíclico dirigido. Analiza tu algoritmo.

### 24.3 Algoritmo de Dijkstra

El algoritmo de Dijkstra resuelve el problema de los caminos más cortos de fuente única en un gráfico dirigido y ponderado  $G = \langle V, E \rangle$  para el caso en que todos los pesos de los bordes son no negativos. En esta sección, por lo tanto, asumimos que  $w_{uv} \geq 0$  para cada arista  $(u, v) \in E$ . Como veremos, con una buena implementación, el tiempo de ejecución del algoritmo de Dijkstra es menor que el del algoritmo de Bellman-Ford.

El algoritmo de Dijkstra mantiene un conjunto  $S$  de vértices cuyos pesos finales del camino más corto de la fuente  $s$  ya han sido determinados. El algoritmo selecciona repetidamente el vértice  $u \in V \setminus S$  con la estimación mínima del camino más corto, agrega  $u$  a  $S$  y relaja todos los bordes dejando  $u$ . En la siguiente implementación, usamos una cola  $Q$  de vértices de prioridad mínima, codificada por sus valores  $d$ .

```

DIJKSTRA.G; w; s/ 1
INICIALIZAR-UNA-FUENTE.G; s/
2 DE;
3 QDG:V 4
mientras Q ≠ ; u D
    EXTRACT-MIN.Q/ 5 SDS
    ⍷fug para cada vértice 2
    7     G:Adj[u] RELAX.u; ; c/
    8

```

El algoritmo de Dijkstra relaja los bordes como se muestra en la Figura 24.6. La línea 1 inicializa los valores de  $d$  y de la forma habitual, y la línea 2 inicializa el conjunto  $S$  al conjunto vacío. El algoritmo mantiene el invariante  $Q \cup S = V$  al comienzo de cada iteración del bucle while de las líneas 4 a 8. La línea 3 inicializa la cola de prioridad mínima  $Q$  para contener todos los vértices en  $V$ ; desde SD; en ese momento, el invariante es verdadero después de la línea 3. Cada vez que pasa por el bucle while de las líneas 4–8, la línea 5 extrae un vértice  $u$  de  $Q$  y la línea 6 lo agrega al conjunto  $S$ , manteniendo así el invariante. (La primera vez que pasa por este bucle,  $u = s$ .) El vértice  $u$ , por lo tanto, tiene la estimación de ruta más corta más pequeña de cualquier vértice en  $V \setminus S$ . Luego, las líneas 7 y 8 relajan cada borde  $(u, v)$  dejando  $u$ , actualizando así la estimación  $d_v$  y el predecesor  $:v$  si podemos mejorar el camino más corto hasta ahora al pasar por  $u$ . Observe que el algoritmo nunca inserta vértices en  $Q$  después de la línea 3 y que cada vértice se extrae de  $Q$ .

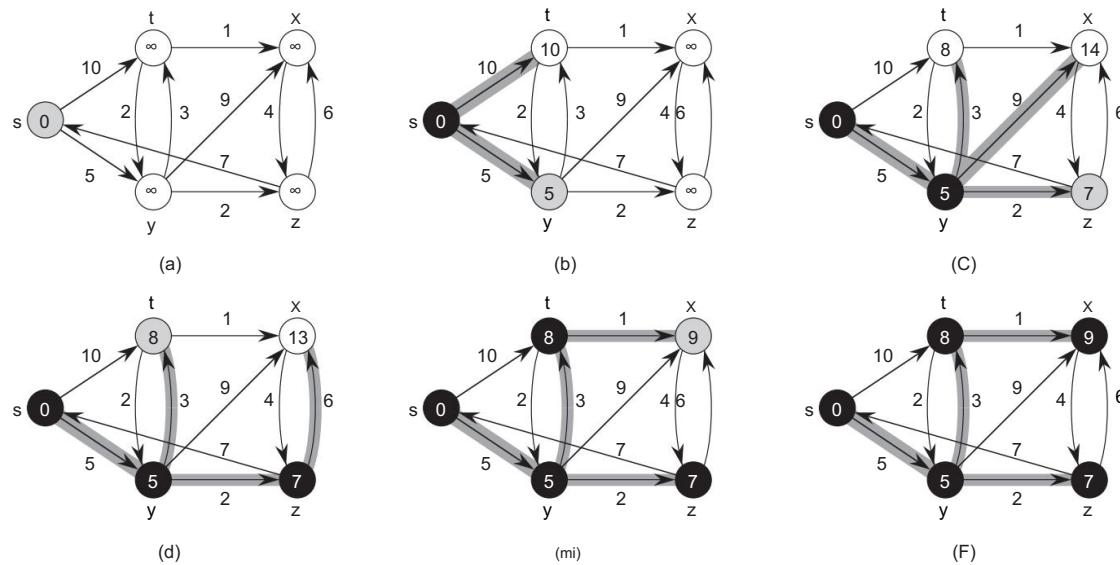


Figura 24.6 La ejecución del algoritmo de Dijkstra. La fuente s es el vértice más a la izquierda. Las estimaciones de la ruta más corta aparecen dentro de los vértices y los bordes sombreados indican valores predecesores.

Los vértices negros están en el conjunto  $S$  y los vértices blancos están en la cola de prioridad mínima QDV  $S$ . (a) La situación justo antes de la primera iteración del ciclo while de las líneas 4–8. El vértice sombreado tiene el valor mínimo  $d$  y se elige como vértice  $u$  en la línea 5. (b)–(f) La situación después de cada iteración sucesiva del bucle while . El vértice sombreado en cada parte se elige como vértice  $u$  en la línea 5 de la siguiente iteración.

Los valores  $d$  y los predecesores que se muestran en la parte (f) son los valores finales.

y se suma a  $S$  exactamente una vez, de modo que el ciclo while de las líneas 4–8 itera exactamente  $|V| - 1$  veces.

Debido a que el algoritmo de Dijkstra siempre elige el vértice "más ligero" o "más cercano" en  $V \setminus S$  para agregarlo al conjunto  $S$ , decimos que usa una estrategia codicosa. El capítulo 16 explica en detalle las estrategias codicosas, pero no es necesario haber leído ese capítulo para comprender el algoritmo de Dijkstra. Las estrategias codicosas no siempre producen resultados óptimos en general, pero como muestran el siguiente teorema y su corolario, el algoritmo de Dijkstra sí calcula los caminos más cortos. La clave es demostrar que cada vez que se suma un vértice  $u$  al conjunto  $S$ , tenemos  $u : d \leq D_{i.s} ; tu$ .

#### Teorema 24.6 (Corrección del algoritmo de Dijkstra)

el algoritmo de Dijkstra, ejecutado en un gráfico dirigido y ponderado  $G = (V; E)$  con función de ponderación no negativa  $w$  y fuente  $s$ , termina en  $u : d \leq D_{i.s} ; u$  para todos los vértices  $u \in V$ .

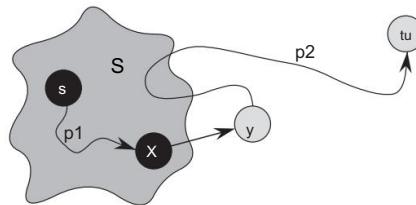


Figura 24.7 La demostración del Teorema 24.6. El conjunto  $S$  no está vacío justo antes de que se le agregue el vértice  $u$ . Nosotros descomponga un camino más corto  $p$  desde la fuente  $s$  hasta el vértice  $u$  en  $s \xrightarrow{p_1} u$ , donde  $u \notin S$ , la primera  $x \neq u$  vértice en el  $p$  no está en  $S$  y  $x \in S$  precede inmediatamente a  $y$ . Los vértices  $x$  e  $y$  son distintos, pero podemos tener  $s \xrightarrow{D} x \xrightarrow{D} u$ . El camino  $p_2$  puede o no reingresar al conjunto  $S$ .

Prueba Usamos el siguiente bucle invariante:

Al comienzo de cada iteración del ciclo while de las líneas 4–8,  $:d D \ i.s; /$   
por cada vértice  $2 \leq V$ .

Basta mostrar para cada vértice  $u \leq 2 \leq V$ , tenemos  $u:d D \ i.s; u/$  en el momento en que  $u$  se agrega al conjunto  $S$ . Una vez que mostramos que  $u:d D \ i.s; u/$ , nos basamos en la propiedad del límite superior para mostrar que la igualdad se mantiene en todo momento a partir de entonces.

Inicialización: Inicialmente,  $SD ;$ , por lo que el invariante es trivialmente verdadero.

Mantenimiento: Queremos mostrar que en cada iteración,  $u:d D \ i.s; u/$  para el vértice añadido al conjunto  $S$ . A efectos de contradicción, sea  $u$  el primer vértice para el que  $u:d \neq i.s; u/$  cuando se suma al conjunto  $S$ . Centraremos nuestra atención en la situación al comienzo de la iteración del bucle while en el que  $u$  se suma a  $S$  y derivamos la contradicción de que  $u:d D \ i.s; u/$  en ese momento examinando el camino más corto de  $s$  a  $u$ . Debemos tener  $u \neq s$  porque  $s$  es el primer vértice que se suma al conjunto  $S$  y  $s:d D \ i.s; s/ D 0$  en ese momento. Porque  $u \neq s$ , también tenemos que  $S \neq ;$  justo antes de que  $u$  se agregue a  $S$ . Debe haber algún camino de  $s$  a  $u$ , porque de lo contrario  $u:d D \ i.s; u/ D 1$  por la propiedad de no trayectoria, lo que violaría nuestra suposición de que  $u:d \neq i.s; u/$ . Como hay al menos un camino, hay un camino más corto  $p$  de  $s$  a  $u$ . Antes de agregar  $u$  a  $S$ , la ruta  $p$  conecta un vértice en  $S$ , a saber,  $s$ , con un vértice en  $VS$ , a saber,  $u$ . Consideraremos el primer vértice  $y$  a lo largo de  $p$  tal que  $y \in VS$ , y sea  $x \in S$  el predecesor de  $y$  a lo largo de  $p$ . Así, como ilustra la figura 24.7, podemos descomponer la ruta  $p$  en  $s$

$\xrightarrow{p_1}$   $bordes.) \xrightarrow{p_2} tu$  (Cualquiera de los caminos  $p_1$  o  $p_2$  puede no tener

Decimos que  $y:d D \ i.s; y/$  cuando  $u$  se suma a  $S$ . Para probar esta afirmación, observe que  $x \in S$ . Entonces, debido a que elegimos  $u$  como el primer vértice para el cual  $u:d \neq i.s; u/$  cuando se suma a  $S$ , se tiene  $x:d D \ i.s; x/$  cuando se agregó  $x$

a S. Borde  $.x; y/$  se relajó en ese momento, y la afirmación se deriva de la propiedad de convergencia.

Ahora podemos obtener una contradicción para probar que  $u:d D \leq s; tu/$ . Como y aparece antes que u en el camino más corto de s a u y todos los pesos de los bordes no son negativos (especialmente los del camino p2), tenemos  $\leq s; y/ \leq s; u/$ , y por lo tanto

$$y:d D \leq s; y/ \leq s; tu/ \quad (24.2)$$

$tu:$

(por la propiedad de límite superior).

Pero debido a que ambos vértices u e y estaban en VS cuando se eligió u en la línea 5, tenemos  $u:d y:d$ . Así, las dos desigualdades en (24.2) son de hecho igualdades, dando

$$y:d D \leq s; y/ D \leq s; tu/ tu: d :$$

En consecuencia,  $u:d D \leq s; u/$ , lo que contradice nuestra elección de u. Concluimos que  $u:d D \leq s; u/$  cuando u se suma a S, y que esta igualdad se mantiene en todo momento a partir de entonces.

Terminación: Al terminar, QD ; que, junto con nuestra invariante anterior QDVS, implica que SDV . Así,  $u:d D \leq s; u/$  para todos los vértices u  $\in V$ . ■

#### Corolario 24.7 Si

ejecutamos el algoritmo de Dijkstra en un grafo dirigido ponderado GD  $.V; E/$  con función de ponderación no negativa w y fuente s, luego, al final, el subgrafo predecesor G es un árbol de caminos más cortos con raíz en s.

Demostración inmediata del teorema 24.6 y la propiedad del subgrafo predecesor. ■

#### Análisis

¿Qué tan rápido es el algoritmo de Dijkstra? Mantiene la cola de prioridad mínima Q llamando a tres operaciones de cola de prioridad: INSERT (implícita en la línea 3), EXTRACT-MIN (línea 5) y DECREASE-KEY (implícita en RELAX, que se llama en la línea 8). El algoritmo llama tanto a INSERT como a EXTRACT-MIN una vez por vértice. Como cada vértice u  $\in V$  se suma al conjunto S exactamente una vez, cada arista en la lista de adyacencia  $Adj[u]$  se examina en el ciclo for de las líneas 7–8 exactamente una vez durante el transcurso del algoritmo. Dado que el número total de aristas en todas las listas de adyacencia es  $|E|$ , este bucle for itera un total de  $|E|$  veces y, por lo tanto, el algoritmo llama a DECREASE-KEY como máximo  $|E|$  veces en total. (Observe una vez más que estamos utilizando un análisis agregado).

El tiempo de ejecución del algoritmo de Dijkstra depende de cómo implementemos la cola de prioridad mínima. Considere primero el caso en el que mantenemos la prioridad mínima

hacer cola aprovechando que los vértices están numerados del 1 al  $jV j$ . Simplemente almacenamos :d en la enésima entrada de una matriz. Cada operación INSERT y DECREASE-KEY toma  $O.1/$  tiempo, y cada operación EXTRACT-MIN toma  $OV /$  tiempo (dado que tenemos que buscar en todo el arreglo), de  $OV$  Si el gráfico es lo suficientemente escaso, en particular,  $ED oV 2 = \lg V^2$  por un tiempo total

$V /$ , podemos mejorar el algoritmo implementando la cola de prioridad mínima con un montón mínimo binario. (Como se discutió en la Sección 6.5, la implementación debe asegurarse de que los vértices y los elementos del montículo correspondientes se mantengan identificadores entre sí). Cada operación EXTRACT-MIN toma tiempo  $O.\lg V /$ . Como antes, hay  $jV j$  tales operaciones. El tiempo para construir el montón mínimo binario es  $OV /$ . Cada operación de DECREASE-KEY toma un tiempo  $O.\lg V /$ , y todavía hay como mucho  $jEj$  tales operaciones. Por lo tanto, el tiempo de ejecución total es  $O..VCE/\lg V /$ , que es  $OE \lg V /$  si todos los vértices son accesibles desde la fuente. Este tiempo de ejecución mejora la implementación sencilla de  $OV 2/-$ tiempo si  $ED oV 2 = \lg V /$ .

De hecho, podemos lograr un tiempo de ejecución de  $OV \lg VCE/$  implementando la cola de prioridad mínima con un montón de Fibonacci (consulte el Capítulo 19). El costo amortizado de cada una de las operaciones  $jV j$  EXTRACT-MIN es de  $O.\lg V /$ , y cada llamada de DECREASE KEY , de las cuales hay como máximo  $jEj$ , toma solo  $O.1/$  del tiempo amortizado. Históricamente, el desarrollo de los montones de Fibonacci estuvo motivado por la observación de que el algoritmo de Dijkstra normalmente realiza muchas más llamadas de DECREASE-KEY que llamadas de EXTRACT-MIN , de modo que cualquier método para reducir el tiempo amortizado de cada operación de DECREASE-KEY a  $O.\lg V /$  sin aumentar el tiempo amortizado de EXTRACT-MIN produciría una implementación asintóticamente más rápida que con montones binarios.

El algoritmo de Dijkstra se parece tanto a la búsqueda en anchura (consulte la Sección 22.2) como al algoritmo de Prim para calcular árboles de expansión mínimos (consulte la Sección 23.2). Es como la búsqueda primero en anchura en que el conjunto S corresponde al conjunto de vértices negros en una búsqueda primero en anchura; Así como los vértices en S tienen sus pesos finales de camino más corto, los vértices negros en una búsqueda primero en anchura tienen sus distancias correctas en anchura.

El algoritmo de Dijkstra es como el algoritmo de Prim en que ambos algoritmos usan una cola de prioridad mínima para encontrar el vértice "más ligero" fuera de un conjunto dado (el conjunto S en el algoritmo de Dijkstra y el árbol que crece en el algoritmo de Prim), agrega este vértice al conjunto, y ajuste los pesos de los vértices restantes fuera del conjunto en consecuencia.

## Ejercicios

### 24.3-1

Ejecute el algoritmo de Dijkstra en el gráfico dirigido de la figura 24.2, primero usando el vértice s como fuente y luego el vértice ' como fuente. Al estilo de la figura 24.6, muestre los valores de dy los vértices en el conjunto S después de cada iteración del bucle while .

## 24.3-2

Dé un ejemplo simple de un gráfico dirigido con bordes de peso negativo para el cual el algoritmo de Dijkstra produce respuestas incorrectas. ¿Por qué no se cumple la prueba del Teorema 24.6 cuando se permiten bordes de peso negativo?

## 24.3-3

Supongamos que cambiamos la línea 4 del algoritmo de Dijkstra por la siguiente.

4 mientras  $jQj > 1$

Este cambio hace que el bucle while ejecute  $jV j$  este 1 veces en lugar de  $jV j$  veces. Es algoritmo propuesto, ¿correcto?

## 24.3-4

El profesor Gaedel ha escrito un programa que, según afirma, implementa el algoritmo de Dijkstra. El programa produce :d y : para cada vértice  $2 V$ . Proporcione un algoritmo OV CE-time para verificar la salida del programa del profesor. Debería determinar si los atributos d y coinciden con los de algún árbol de caminos más cortos.

Puede suponer que todos los pesos de los bordes son no negativos.

## 24.3-5

El profesor Newman cree que ha elaborado una prueba más sencilla de la corrección del algoritmo de Dijkstra. Afirma que el algoritmo de Dijkstra relaja los bordes de cada ruta más corta en el gráfico en el orden en que aparecen en la ruta y, por lo tanto, la propiedad de relajación de la ruta se aplica a cada vértice accesible desde la fuente. Muestre que el profesor se equivoca al construir un gráfico dirigido para el cual el algoritmo de Dijkstra podría relajar los bordes de un camino más corto fuera de orden.

## 24.3-6

Nos dan un grafo dirigido  $GD .V; E/$  en el que cada borde  $.u; / 2 E$  tiene un valor asociado  $ru; /$ , que es un número real en el rango  $0 ru; / 1$  que representa la confiabilidad de un canal de comunicación desde el vértice u hasta el vértice .

Interpretamos  $ru; /$  como la probabilidad de que el canal de u a no falle, y suponemos que estas probabilidades son independientes. Proporcione un algoritmo eficiente para encontrar el camino más confiable entre dos vértices dados.

## 24.3-7

Sea  $GD .V; E/$  sea un gráfico dirigido y ponderado con una función de peso positiva w WE ! f1; 2; : : : ; W g para algún número entero positivo W tices tienen los mismos, y supongamos que no hay dos ver pesos de ruta más corta desde los vértices de origen. Supongamos ahora que definimos un grafo dirigido no ponderado  $G0 D .V [ E0 /$  reemplazando cada arista  $.u; / 2 E$  con  $wu; /$  filos de peso unitario en serie. ¿Cuántos vértices tiene  $G0$ ? Ahora suponga que ejecutamos una búsqueda primero en amplitud en  $G0$ . Muestra esa

el orden en el que la búsqueda en anchura de G0 colorea los vértices en V negro es el mismo orden en el que el algoritmo de Dijkstra extrae los vértices de V de la cola de prioridad cuando se ejecuta en G.

#### 24.3-8

Sea GD .V; E/ sea un gráfico dirigido y ponderado con una función de peso no negativa w WE ! f0; 1; : : : ; W g para algún número entero no negativo W . Modifique el algoritmo de Dijkstra para calcular las rutas más cortas desde un vértice de origen dado en OW VCE/ tiempo.

#### 24.3-9

Modifique su algoritmo del ejercicio 24.3-8 para que se ejecute en O..VCE/lg W / time. (Pista: ¿cuántas estimaciones distintas del camino más corto puede haber en VS en cualquier momento?)

#### 24.3-10

Suponga que se nos da un gráfico dirigido y ponderado GD .V; E/ en el que los bordes que salen del vértice de origen s pueden tener pesos negativos, todos los demás pesos de los bordes no son negativos y no hay ciclos de peso negativo. Argumente que el algoritmo de Dijkstra encuentra correctamente los caminos más cortos desde s en este gráfico.

### 24.4 Restricciones de diferencia y caminos más cortos

El capítulo 29 estudia el problema general de programación lineal, en el que deseamos optimizar una función lineal sujeta a un conjunto de desigualdades lineales. En esta sección, investigamos un caso especial de programación lineal que reducimos a encontrar las rutas más cortas desde una sola fuente. Entonces podemos resolver el problema de los caminos más cortos de fuente única que resulta al ejecutar el algoritmo Bellman-Ford, resolviendo así también el problema de programación lineal.

#### Programación lineal

En el problema general de programación lineal, tenemos una matriz mn A, un vector m b y un vector n c. Deseamos encontrar un vector x de n elementos que maximice la función objetivo Pn iD1 cixi sujeta a las m restricciones dadas por Ax b.

Aunque el algoritmo simplex, que es el tema central del capítulo 29, no siempre se ejecuta en tiempo polinomial en el tamaño de su entrada, existen otros algoritmos de programación lineal que sí se ejecutan en tiempo polinomial. Ofrecemos aquí dos razones para comprender la configuración de los problemas de programación lineal. En primer lugar, si sabemos que

puede formular un problema dado como un problema de programación lineal de tamaño polinomial, inmediatamente tenemos un algoritmo de tiempo polinomial para resolver el problema. En segundo lugar, existen algoritmos más rápidos para muchos casos especiales de programación lineal. Por ejemplo, el problema del camino más corto de un solo par (ejercicio 24.4-4) y el problema del flujo máximo (ejercicio 26.1-5) son casos especiales de programación lineal.

A veces no nos importa realmente la función objetivo; simplemente deseamos encontrar cualquier solución factible, es decir, cualquier vector  $x$  que satisfaga  $Ax \leq b$ , o determinar que no existe ninguna solución factible. Nos centraremos en uno de esos problemas de viabilidad.

#### Sistemas de restricciones de diferencias

En un sistema de restricciones de diferencia, cada fila de la matriz de programación lineal  $A$  contiene un 1 y un 1, y todas las demás entradas de  $A$  son 0. Por lo tanto, las restricciones dadas por  $Ax \leq b$  son un conjunto de  $m$  restricciones de diferencia que involucran  $n$  incógnitas, en la que cada restricción es una desigualdad lineal simple de la forma

$$x_j - x_i \leq b_k;$$

donde  $1 \leq i < j \leq n$ ,  $1 \leq k \leq m$ .

Por ejemplo, considere el problema de encontrar un 5-vector  $x = [x_1, x_2, x_3, x_4, x_5]$  que satisfaga

$$\begin{array}{rcccl} 1 & 1000 & 1000 & 1 & 0 \\ & & & & 1 \\ 0100 & 1 & 10100 & & 1 \\ & & & x_2 & 5 \\ 10010 & & & x_3 & 4 \\ 00110 & & & x_4 & 1 \\ 00101 & & & x_5 & 3 \\ 0001 & & & 1 & \sim \\ & & & x_1 & 3 & \sim \end{array}$$

Este problema es equivalente a encontrar valores para las incógnitas  $x_1, x_2, x_3, x_4, x_5$ , satisfaciendo las siguientes 8 restricciones de diferencia:

$$x_1 - x_2 \leq 0, \quad (24.3)$$

$$x_1 - x_5 \leq 1, \quad (24.4)$$

$$x_2 - x_5 \leq 1, \quad (24.5)$$

$$x_3 - x_1 \leq 5, \quad (24.6)$$

$$x_4 - x_1 \leq 4, \quad (24.7)$$

$$x_4 - x_3 \leq 1, \quad (24.8)$$

$$x_5 - x_3 \leq 3, \quad (24.9)$$

$$x_5 - x_4 \leq 3. \quad (24.10)$$

Una solución a este problema es  $x \in D .5; 3; 0; 1; 4/$ , que puedes comprobar directamente comprobando cada desigualdad. De hecho, este problema tiene más de una solución.

Otro es  $x_0 \in D .0; 2; 5; 4; 1/$ . Estas dos soluciones están relacionadas: cada componente de  $x_0$  es 5 veces mayor que el componente correspondiente de  $x$ . Este hecho no es mera coincidencia.

#### Lema 24.8

Sea  $x \in D .x_1; x_2; \dots; x_n/$  sea una solución de un sistema  $Ax = b$  de restricciones en diferencia, y sea  $d$  una constante cualquiera. Entonces  $x + Cd \in D .x_1 + C_d; x_2 + C_d; \dots; x_n + C_d/$  es también una solución para  $Ax = b$ .

Prueba Para cada  $x_i$  y  $x_j$ , tenemos  $x_j \in C_d /$  . $x_i \in C_d / D x_j \leq x_i$ . Así, si  $x$  satisface  $Ax = b$ , lo mismo ocurre con  $x + C_d$ . ■

Los sistemas de restricciones de diferencia ocurren en muchas aplicaciones diferentes. Por ejemplo, las incógnitas  $x_i$  pueden ser tiempos en los que ocurrirán los eventos. Cada restricción establece que debe transcurrir al menos una cierta cantidad de tiempo, o como máximo una cierta cantidad de tiempo, entre dos eventos. Tal vez los eventos sean trabajos a realizar durante el montaje de un producto. Si aplicamos un adhesivo que tarda 2 horas en fraguar en el tiempo  $x_1$  y tenemos que esperar hasta que fragüe para instalar una pieza en el tiempo  $x_2$ , entonces tenemos la restricción de que  $x_2 \geq x_1 + 2$ , de manera equivalente, que  $x_1 \leq x_2 - 2$ . Alternativamente, es posible que necesitemos que la pieza se instale después de que se haya aplicado el adhesivo, pero a más tardar cuando el adhesivo se haya fraguado a la mitad. En este caso, obtenemos el par de restricciones  $x_2 \geq x_1$  y  $x_2 \leq x_1 + 1$ , de manera equivalente,  $x_1 \leq x_2 - 1$  y  $x_2 \leq x_1 + 1$ .

#### Gráficos de restricciones

Podemos interpretar sistemas de restricciones de diferencia desde un punto de vista teórico de grafos. En un sistema  $Ax = b$  de restricciones de diferencia, vemos la matriz  $A$  de programación lineal  $mn$  como la transpuesta de una matriz de incidencia (vea el ejercicio 22.1-7) para una gráfica con  $n$  vértices y  $m$  aristas. Cada vértice  $i$  en el gráfico, para  $i \in D 1; 2; \dots; n$ , corresponde a una de las  $n$  variables desconocidas  $x_i$ . Cada arista dirigida en el gráfico corresponde a una de las  $m$  desigualdades que involucran dos incógnitas.

Más formalmente, dado un sistema  $Ax = b$  de restricciones de diferencia, el correspondiente grafo de restricciones de ing es un grafo dirigido y ponderado  $G = (V, E)$ , donde

$V = D f_0; 1; \dots; n_g$

y

$E = \{ (f_i, j) \mid W x_j \leq b_k \text{ es una restricción } [f_0; 1; \dots; 0; 2; \dots; 0; 3; \dots; 0; n_g]$

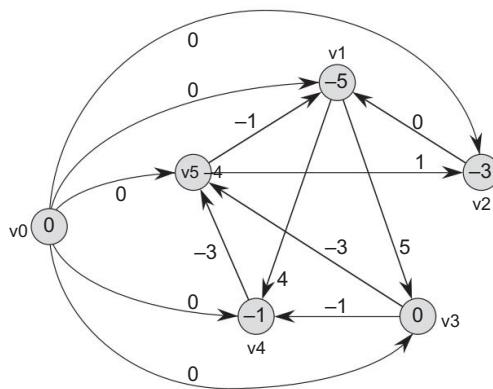


Figura 24.8 El gráfico de restricciones correspondiente al sistema (24.3)–(24.10) de restricciones de diferencia. El valor de  $x_i$  aparece en cada vértice  $i$ . Una solución factible del sistema es  $x = [0; 5; 3; 0; 1; 4]$ .

El grafo de restricciones contiene el vértice adicional 0, como veremos en breve, para garantizar que el grafo tenga algún vértice que pueda alcanzar a todos los demás vértices. Por lo tanto, el conjunto de vértices  $V$  consta de un vértice  $i$  para cada  $x_i$  desconocido, más un adicional. El vértice 0. Conjunto de aristas  $E$  contiene una arista para cada restricción de diferencia, más una arista  $(0; i)$  para cada  $x_i$  desconocida . Si  $x_j \neq b_k$  es una restricción de diferencia, entonces el peso de la arista  $(i; j)$  es  $w_{ij}$ . El peso de cada arista que sale de 0 es 0. La figura 24.8 muestra el gráfico de restricciones para el sistema (24.3)–(24.10) de restricciones de diferencia.

El siguiente teorema muestra que podemos encontrar una solución a un sistema de restricciones de diferencia al encontrar los pesos de la ruta más corta en el gráfico de restricción correspondiente.

#### Teorema 24.9

Dado un sistema  $Ax = b$  de restricciones diferenciales, sea  $G = (V; E)$  sea el gráfico de restricción correspondiente. Si  $G$  no contiene ciclos de peso negativo, entonces

$$x = [0; 1; 2; 3; \dots; n] \text{ es una solución factible para el sistema.} \quad (24.11)$$

es una solución factible para el sistema. Si  $G$  contiene un ciclo de peso negativo, entonces no hay una solución factible para el sistema.

**Demostración** Primero mostramos que si el gráfico de restricciones no contiene ciclos de peso negativo, entonces la ecuación (24.11) da una solución factible. Considere cualquier borde  $(i; j) \in E$ . Por la desigualdad del triángulo,  $|x_i - x_j| \leq w_{ij}$ , de manera equivalente,  $|x_i - x_j| \leq |w_{ij}|$ . Así, dejando  $x_i = 0$  y

$x_j \leq 1.0; j \neq i$  satisface la restricción de diferencia  $x_j - x_i \leq w_{ij}; j \neq i$  que corresponde a la arista  $i \rightarrow j$ .

Ahora mostramos que si el gráfico de restricciones contiene un ciclo de peso negativo, entonces el sistema de restricciones de diferencia no tiene una solución factible. Sin pérdida de generalidad, sea el ciclo de peso negativo  $c: h_1, h_2, \dots, h_k$ , donde ( $\text{El vértice corresponde a las siguientes restricciones}$ )  $0 \leq x_i \leq 1$  para  $i = 1, \dots, k$ . No puede estar en el ciclo  $c$ , porque no tiene aristas de entrada.) Ciclo  $c$  de diferencia:

$$\begin{array}{ll} x_2 - x_1 & \leq w_{12}; \\ x_3 - x_2 & \leq w_{23}; \\ & \vdots \\ x_k - x_1 & \leq w_{1k}; \\ x_1 - x_k & \leq w_{kk}; \end{array}$$

Supondremos que  $x$  tiene una solución que satisface cada una de estas  $k$  desigualdades y luego derivaremos una contradicción. La solución también debe satisfacer la desigualdad que resulta cuando sumamos las  $k$  desigualdades. Si sumamos los lados izquierdos, cada  $x_i$  desconocido se suma una vez y se resta una vez (recuerde que eso implica  $x_1 \leq x_k$ ), de modo que el lado izquierdo de la suma es 0. El lado derecho suma  $w_{kk}$ , y así obtenemos  $0 \leq w_{kk}$ . Pero como  $c$  es un ciclo de peso negativo,  $w_{kk} < 0$ , y obtenemos la contradicción de que  $0 \leq w_{kk} < 0$ . ■

#### Resolver sistemas de restricciones de diferencia

El teorema 24.9 nos dice que podemos usar el algoritmo de Bellman-Ford para resolver un sistema de restricciones en diferencias. Debido a que el gráfico de restricciones contiene aristas desde el vértice de origen hasta todos los demás vértices, se puede acceder a cualquier ciclo de peso negativo en el gráfico de restricciones desde Si el algoritmo de Bellman-Ford devuelve VERDADERO, entonces los pesos de la ruta más corta dan una solución factible al sistema. En la figura 24.8, por ejemplo, los pesos del camino más corto proporcionan la solución factible  $x = (0.5, 3, 0, 1, 4)$ , y por el Lema 24.8,  $x$  es una solución factible para cualquier constante  $d$ . Si el algoritmo de Bellman-Ford devuelve FALSO, no hay una solución factible para el sistema de restricciones de diferencia.

Un sistema de restricciones de diferencia con  $m$  restricciones sobre  $n$  incógnitas produce un gráfico con  $n + m$  vértices y  $n + m$  aristas. Así, usando el algoritmo de Bellman-Ford, podemos resolver el sistema en  $O(nm)$  tiempo.

El ejercicio 24.4-5 le pide que modifique el algoritmo para que se ejecute en tiempo  $O(nm)$ , incluso si  $m$  es mucho menor que  $n$ .

**Ejercicios****24.4-1**

Encuentre una solución factible o determine que no existe una solución factible para el siguiente sistema de restricciones en diferencia:

x1 x2	1 ,
x1x4 _	4 ,
x2x3 _	2 ,
x2x5 _	7 ,
x2 x6	5 ,
x3 x6	10 ,
x4 x2	2 ,
x5 x1	1 ,
x5 x4	3 ,
x6 x3	8

**24.4-2**

Encuentre una solución factible o determine que no existe una solución factible para el siguiente sistema de restricciones en diferencia:

x1 x2	4 ,
x1 x5	5 ,
x2 x4	6 ,
x3 x2	1 ,
x4 x1	3 ,
x4 x3	5 ,
x4 x5	10 ,
x5 x3	4 ,
x5 x4	8

**24.4-3**

¿Puede cualquier camino más corto pesar desde el nuevo vértice? Explicar.

**24.4-4**

Exprese el problema del camino más corto de un solo par como un programa lineal.

## 24.4-5

Muestre cómo modificar ligeramente el algoritmo de Bellman-Ford para que cuando lo usemos para resolver un sistema de restricciones en diferencias con  $m$  desigualdades en  $n$  incógnitas, el tiempo de ejecución sea  $O(nm)$ .

## 24.4-6

Suponga que además de un sistema de restricciones de diferencia, queremos manejar restricciones de igualdad de la forma  $x_i \leq x_j \leq b_k$ . Muestre cómo adaptar el algoritmo de Bellman Ford para resolver esta variedad de sistemas de restricciones.

## 24.4-7

Muestre cómo resolver un sistema de restricciones de diferencia mediante un algoritmo similar a Bellman-Ford que se ejecuta en un gráfico de restricciones sin el vértice adicional

## 24.4-8 ?

Sea  $Ax = b$  un sistema de  $m$  restricciones de diferencia en  $n$  incógnitas. Muestre que el algoritmo de Bellman-Ford, cuando se ejecuta en el gráfico de restricción correspondiente, maximiza  $\sum_{i=1}^m x_i$  sujeto a  $Ax = b$  y  $x_i \geq 0$  para todos los  $x_i$ .

## 24.4-9 ?

Muestre que el algoritmo de Bellman-Ford, cuando se ejecuta en el gráfico de restricciones para un sistema  $Ax = b$  de restricciones de diferencia, minimiza la cantidad  $\max f_i \min g_i$  sujeta a  $Ax = b$ . Explique cómo este hecho podría ser útil si el algoritmo se usa para programar trabajos de construcción.

## 24.4-10

Suponga que cada fila en la matriz  $A$  de un programa lineal  $Ax = b$  corresponde a una restricción de diferencia, una restricción de una sola variable de la forma  $x_i \leq b_k$ , o una sola  $b_k$ . Muestre cómo adaptar la restricción variable de Bellman-Ford del algoritmo de sistemas de restricciones.

## 24.4-11

Proporcione un algoritmo eficiente para resolver un sistema  $Ax = b$  de restricciones de diferencia cuando todos los elementos de  $b$  tienen valores reales y todas las incógnitas  $x_i$  deben ser números enteros.

## 24.4-12 ?

Proporcione un algoritmo eficiente para resolver un sistema  $Ax = b$  de restricciones de diferencia cuando todos los elementos de  $b$  tienen valores reales y un subconjunto específico de algunas, pero no necesariamente todas, las incógnitas  $x_i$  deben ser números enteros.

## 24.5 Pruebas de las propiedades de los caminos más cortos

A lo largo de este capítulo, nuestros argumentos de corrección se han basado en la desigualdad del triángulo, la propiedad del límite superior, la propiedad de no trayectoria, la propiedad de convergencia, la propiedad de relajación de la trayectoria y la propiedad del subgrafo predecesor. Enunciamos estas propiedades sin demostración al comienzo de este capítulo. En esta sección, los demostramos.

### La desigualdad triangular

Al estudiar la búsqueda primero en anchura (Sección 22.2), demostramos como el Lema 22.1 una propiedad simple de las distancias más cortas en grafos no ponderados. La desigualdad triangular generaliza la propiedad a gráficos ponderados.

#### Lema 24.10 (Desigualdad triangular)

Sea  $G = \langle V, E \rangle$  sea un gráfico dirigido y ponderado con una función de peso  $w : E \rightarrow \mathbb{R}$  y el vértice fuente  $s$ . Luego, para todas las aristas  $u \in E$ , tenemos

$es \leq su \leq Cwu$

Prueba Suponga que  $p$  es el camino más corto desde la fuente  $s$  hasta el vértice. Entonces  $p$  no tiene más peso que cualquier otro camino de  $s$  a  $t$ . Específicamente, el camino  $p$  no tiene más peso que el camino particular que toma el camino más corto desde la fuente  $s$  hasta el vértice  $u$  y luego toma el borde  $u \rightarrow t$ .

El ejercicio 24.5-3 le pide que maneje el caso en el que no existe el camino más corto de  $s$  a  $t$ .



### Efectos de la relajación en las estimaciones del camino más corto

El siguiente grupo de lemas describe cómo se ven afectadas las estimaciones de la ruta más corta cuando ejecutamos una secuencia de pasos de relajación en los bordes de un gráfico dirigido ponderado que ha sido inicializado por INITIALIZE-SINGLE-SOURCE.

#### Lema 24.11 (Propiedad de límite superior)

Sea  $G = \langle V, E \rangle$  sea un gráfico dirigido y ponderado con una función de peso  $w : E \rightarrow \mathbb{R}$

Sea  $s \in V$  el vértice fuente, y sea el gráfico inicializado por INITIALIZE-SINGLE-SOURCE( $G, s$ ). Entonces,  $d[i, s] \leq d[i, s]$  para todos los  $v \in V$  mantenidos sobre cualquier secuencia, y esta invariante es de pasos de relajación en los bordes de  $G$ . Además, una vez que  $d$  alcanza su límite inferior  $i, s \leq d[i, s]$ , nunca cambia.

Prueba Probamos el invariante  $:d \leq s; /$  para todos los vértices  $2 \leq V$  por inducción sobre el número de pasos de relajación.

Para la base,  $:d \leq s; /$  es cierto después de la inicialización, ya que  $:d \leq D_1$  implica  $:d \leq s; /$  para todos los  $2 \leq V$  fsg, y dado que  $s \leq D_0 \leq s; /$  (nótese que  $\leq s; / \leq D_1$  si  $s$  está en un ciclo de ponderación negativa y 0 en caso contrario).

Para el paso inductivo, considere la relajación de un borde  $.u; /$ . Por la hipótesis inductiva,  $x \leq s; /$  para todo  $x \leq V$  antes de la relajación. El único valor de  $d$  que puede cambiar es  $:d$ . Si cambia, tenemos

$:d \leq t \leq d \leq C \leq w; /$

$es; / \leq C \leq w; /$  (por la hipótesis inductiva)  $\leq s; /$  (por la desigualdad triangular),

y así se mantiene el invariante.

Para ver que el valor de  $:d$  nunca cambia una vez  $:d \leq s; /$ , tenga en cuenta que habiendo alcanzado su límite inferior,  $:d$  no puede disminuir porque acabamos de demostrar que  $:d \leq s; /$ , y no puede aumentar porque los pasos de relajación no aumentan  $d$  valores. ■

#### Corolario 24.12 (Propiedad de no camino)

Suponga que en un grafo dirigido ponderado  $GD .V; E/$  con función de peso  $w : WE \rightarrow R$ , ningún camino conecta un vértice fuente  $s \leq V$  a un vértice dado  $2 \leq V$ .

Luego, después de inicializar el gráfico con  $INITIALIZE-SINGLE-SOURCE.G; s/$ , tenemos  $:d \leq s; / \leq D_1$ , y esta igualdad se mantiene como un invariante sobre cualquier secuencia de pasos de relajación en los bordes de  $G$ .

Prueba Por la propiedad del límite superior, siempre tenemos  $1 \leq D \leq s; /$  así  $:d \leq D \leq s; /$ , y  $:d \leq D \leq s; /$ . ■

#### Lema 24.13

Sea  $GD .V; E/$  sea un gráfico dirigido y ponderado con una función de peso  $w : WE \rightarrow R$ , y sea  $.u; / \leq E$ . Luego, inmediatamente después de relajar el borde  $.u; /$  ejecutando  $RELAX.u; /$ , tenemos  $:d \leq u \leq C \leq w; /$ .

Prueba Si, justo antes de relajar el borde  $.u; /$ , tenemos  $:d > u \leq C \leq w; /$ , luego  $:d \leq D \leq C \leq w; /$  después. Si, en cambio,  $:d \leq u \leq C \leq w; /$  Justo antes de la relajación, entonces ni  $u \leq d$  ni  $:d$  cambian, y así  $:d \leq u \leq C \leq w; /$  después. ■

#### Lema 24.14 (Propiedad de convergencia)

Sea  $GD .V; E/$  sea un gráfico dirigido y ponderado con una función de peso  $w : WE \rightarrow R$ , sea  $s \leq V$  un vértice fuente y sea  $s \neq u$ . Sea un camino más corto en  $G$  para

algunos vértices  $u; 2$  voltios Supongamos que  $G$  se inicializa con INITIALIZE-SINGLE SOURCE.G;  $s/$  y luego una secuencia de pasos de relajación que incluye la llamada RELAX. $u; ; w/$  se ejecuta en las aristas de  $G$ . Si  $u:d D i.s; u/$  en cualquier momento antes de la llamada, luego  $:d D i.s; /$  en todo momento después de la llamada.

Demostración Por la propiedad del límite superior, si  $u:d D i.s; u/$  en algún momento antes de relajar el borde  $.u; /$ , entonces esta igualdad se mantiene a partir de entonces. En particular, después de relajar el borde  $.u; /$ , tenemos

$$:d \quad u:d C wu; / D i.s; \quad (\text{por el Lema 24.13})$$

$$u/ C wu; / D i.s; /$$

(por el Lema 24.1) .

Por la propiedad del límite superior,  $:d D i.s; /$ , de lo que concluimos que  $:d D i.s; /$ , y esta igualdad se mantiene a partir de entonces. ■

#### Lema 24.15 (Propiedad de relajación del camino)

Sea  $GD .V; E/$  sea un gráfico dirigido y ponderado con una función de peso  $w$   $WE \rightarrow R$ , y sea  $s \in V$  un vértice fuente. Considere cualquier camino más corto  $p$   $D h_0; 1; \dots; k$  de  $s \in G$  se inicializa mediante INITIALIZE-SINGLE SOURCE.G;  $s/$  ya  $k$ . luego ocurre una secuencia de pasos de relajación que en orden, relajar los bordes  $.0; 1/; .1; 2/; \dots; .k; k/$ , entonces  $k:d D i.s; k/$  después de estas relajaciones y en todo momento después. Esta propiedad se mantiene sin importar qué otras relajaciones de borde ocurran, incluidas las relajaciones que se entremezclan con las relajaciones de los bordes de  $p$ .

Demostración Demostramos por inducción que después de que se relaja el borde  $i$ -ésimo del camino  $p$ , tenemos  $i:d D i.s; i/$ . Para la base,  $i = 0$ , y antes de que se hayan relajado los bordes de  $p$ , tenemos de la inicialización que  $0:d D s:d D 0 D i.s; s/$ . Por la propiedad de límite superior, el valor de  $s:d$  nunca cambia después de la inicialización.

Para el paso inductivo, suponemos que  $i-1:d D i.s; i-1/$ , y examinamos lo que sucede cuando relajamos el borde  $.i; i/$ . Por la propiedad de convergencia, después de relajar este borde, tenemos  $i:d D i.s; i/$ , y esta igualdad se mantiene en todo momento a partir de entonces. ■

#### Relajación y árboles de caminos más cortos

Ahora mostramos que una vez que una secuencia de relajaciones ha causado que las estimaciones del camino más corto converjan a los pesos del camino más corto, el subgrafo predecesor  $G$  inducido por los valores resultantes es un árbol de caminos más cortos para  $G$ . Comenzamos con el siguiente lema , lo que muestra que el subgrafo predecesor siempre forma un árbol con raíz cuya raíz es la fuente.

**Lema 24.16**

Sea  $G$  .V;  $E$  / sea un gráfico dirigido y ponderado con una función de peso  $w$  WE ! R, sea  $s \in V$  un vértice fuente y suponga que  $G$  no contiene ciclos de peso negativo que sean alcanzables desde  $s$ . Luego, después de inicializar el gráfico con INITIALIZE SINGLE-SOURCE.G; s/, el subgrafo predecesor  $G$  forma un árbol enraizado con raíz  $s$ , y cualquier secuencia de pasos de relajación en las aristas de  $G$  mantiene esta propiedad como invariante.

Prueba Inicialmente, el único vértice en  $G$  es el vértice fuente, y el lema es trivialmente verdadero. Considere un subgrafo predecesor  $G$  que surge después de una secuencia de pasos de relajación. Probaremos primero que  $G$  es acíclico. Supongamos, en aras de la contradicción, que algún paso de relajación crea un ciclo en el gráfico  $G$ . Sea el ciclo  $c$   $D h_0; 1; \dots; k_i$ , donde  $k \geq 0$ . Entonces,  $i: f o^r i D 1; 2; \dots; k$  y, sin pérdida de generalidad, podemos suponer que la arista relajante  $.k_1; k$  creó el ciclo en  $G$ .

Afirmamos que todos los vértices del ciclo  $c$  son accesibles desde la fuente  $s$ . ¿Por qué? Cada vértice en  $c$  tiene un predecesor no NIL, por lo que a cada vértice en  $c$  se le asignó una estimación de ruta más corta finita cuando se le asignó su valor no NIL. Por la propiedad del límite superior, cada vértice en el ciclo  $c$  tiene un peso de camino más corto finito, lo que implica que es accesible desde  $s$ .

Examinaremos las estimaciones del camino más corto en  $c$  justo antes de la llamada RELAX. $k_1; k; w$ / y muestran que  $c$  es un ciclo de ponderación negativa, lo que contradice la suposición de que  $G$  no contiene ciclos de ponderación negativa que sean accesibles desde la fuente. Justo antes de la llamada, tenemos para  $i: f o^r i D 1; 2; \dots; k_1; k$ , la última  $2; \dots; d D i_1: d C w. i_1; i$ . Si actualización de  $i: d$  fue por la asignación Así, para  $i: D 1; \dots; i_1: d$  cambió desde entonces, disminuyó. Por lo tanto, justo antes de la llamada RELAX. $k_1; k; w$ /, tenemos

$$i: d \leq i_1: d C w. i_1; i \quad \text{para todo } i: D 1; 2; \dots; k_1 \quad (24.12)$$

Como  $k$ : se cambia por la llamada, inmediatamente antes también tenemos la desigualdad estricta

$$k: d > k_1: d C w. k_1; k : \dots$$

Sumando esta desigualdad estricta con las  $k_1$  desigualdades (24.12), obtenemos la suma de las estimaciones del camino más corto alrededor del ciclo  $c$ :

$$\begin{array}{ccccc} k & & & k \\ X i: d > X .i_1: d C w. i_1; i // & & & & \\ iD_1 & & iD_1 & & \\ & k & & k \\ & & & & \\ DX i_1: d CX w. i_1; i : & & & & \\ & iD_1 & & iD_1 & \end{array}$$

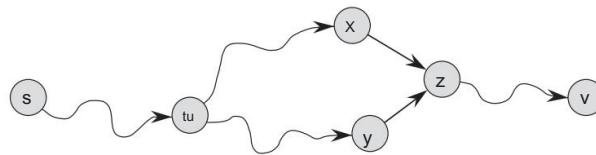


Figura 24.9 Demostración de que un camino simple en  $G$  desde la fuente  $s$  hasta el vértice es único. Si hay dos caminos  $p_1 (s \rightarrow u \rightarrow x \rightarrow \dots)$  y  $p_2 (s \rightarrow u \rightarrow y \rightarrow \dots)$ , donde  $x \neq y$ , entonces  $\sum D(x) > \sum D(y)$ , una contradicción.

Pero

$$\sum_{i=1}^k d(X_i) = \sum_{i=1}^k d(Y_i);$$

ya que cada vértice del ciclo  $c$  aparece exactamente una vez en cada sumatoria. Esta igualdad implica

$$0 > \sum_{i=1}^k w_i;$$

Así, la suma de pesos alrededor del ciclo  $c$  es negativa, lo que proporciona la contradicción deseada.

Ahora hemos probado que  $G$  es un grafo acíclico dirigido. Para demostrar que forma un árbol enraizado con raíz  $s$ , basta (vea el ejercicio B.5-2) para probar que para cada vértice  $v \in V$ , existe un único camino simple de  $s$  a  $v$  en  $G$ .

Primero debemos mostrar que existe un camino desde  $s$  para cada vértice en  $V$ . Los vértices en  $V$  son aquellos con valores distintos de NIL, más  $s$ . La idea aquí es probar por inducción que existe un camino desde  $s$  hasta todos los vértices en  $V$ . Dejamos los detalles como Ejercicio 24.5-6.

Para completar la prueba del lema, ahora debemos demostrar que para cualquier vértice  $v \in V$ , el grafo  $G$  contiene como máximo un camino simple de  $s$  a  $v$ .

Supongamos de otra manera. Es decir, suponga que, como ilustra la figura 24.9,  $G$  contiene dos caminos simples desde  $s$  hasta algún vértice:  $p_1$ , que

$s \rightarrow u \rightarrow x \rightarrow \dots$  donde  $x \neq y$  (aunque  $u \neq y$   $p_2$ , que  $s \rightarrow u \rightarrow y \rightarrow \dots$ , descomponemos en  $s$  descomponemos en  $s \rightarrow u \rightarrow y \rightarrow \dots$  podría ser  $s \rightarrow y \rightarrow \dots$  podría ser  $\dots$ ). Pero entonces,  $\sum D(x) > \sum D(y)$ , lo que implica la contradicción de que  $x \neq y$ . Concluimos que  $G$  contiene un único camino simple de  $s$  a  $v$ . ■

Ahora podemos demostrar que si, después de haber realizado una secuencia de pasos de relajación, a todos los vértices se les han asignado sus verdaderos pesos de camino más corto, entonces el predecesor o subgrafo  $G$  es un árbol de caminos más cortos.

**Lema 24.17 (propiedad del subgrafo predecesor)**

Sea  $G = \langle V, E \rangle$  un gráfico dirigido y ponderado con una función de peso  $w : E \rightarrow \mathbb{R}$ , sea  $s \in V$  un vértice fuente y suponga que  $G$  no contiene ciclos de peso negativo que sean alcanzables desde  $s$ . Llamemos  $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$  y luego ejecutar cualquier secuencia de pasos de relajación en los bordes de  $G$  que produzca  $d$  para todos los  $v \in V$ . Entonces, el subgrafo predecesor  $G'$  es un árbol de caminos más cortos con raíz en  $s$ .

**Prueba** Debemos demostrar que las tres propiedades de los árboles de caminos más cortos dadas en la página 647 se cumplen para  $G'$ . Para mostrar la primera propiedad, debemos demostrar que  $V$  es el conjunto de vértices alcanzables desde  $s$ . Por definición, un peso de camino más corto  $i.s; i/f$  es finito si y solo si es alcanzable desde  $s$ , y por lo tanto los vértices que son alcanzables desde  $s$  son exactamente aquellos con valores finitos de  $d$ . Pero a un vértice  $v \in V$  se le ha asignado un valor finito para  $d$  si y sólo si  $v \neq NIL$ . Así, los vértices en  $V$  son exactamente los alcanzables desde  $s$ .

La segunda propiedad se sigue directamente del Lema 24.16.

Resta, por lo tanto, demostrar la última propiedad de los árboles de caminos más cortos: para cada vértice  $v \in V$ , el único camino simple  $s -> v$  en  $G'$  es el camino más corto de  $s$  a  $v$  en  $G$ . Sea  $p = D[i_0; i_1; \dots; i_k]$ , donde  $i_0 = s$ . Para  $i \in \{1, 2, \dots, k\}$ ,  $D[i]$  tiene ambos  $i:d$  y  $i:f$ . Sumando los pesos a lo largo del camino  $p$  se obtiene

$$\begin{aligned}
 & w.p = w.i_0; i_1 + w.i_1; i_2 + \dots + w.i_{k-1}; i_k \\
 & = w.i_0; i_1 + w.i_1; i_2 + \dots + w.i_{k-1}; i_k \\
 & \leq d[i_0; i_1] + d[i_1; i_2] + \dots + d[i_{k-1}; i_k] \\
 & = d[i_0; i_k] \\
 & = d[s; v]
 \end{aligned}$$

Así,  $w.p = d[s; v]$ . Desde  $i.s; i/f$  es un límite inferior del peso de cualquier camino de  $s$  a  $v$ , concluimos que  $w.p = d[s; v]$ , y por lo tanto  $p$  es el camino más corto de  $s$  a  $v$ .

**Ejercicios****24.5-1**

Proporcione dos árboles de caminos más cortos para el gráfico dirigido de la figura 24.2 (en la página 648) que no sean los dos que se muestran.

## 24.5-2

Dé un ejemplo de un gráfico dirigido ponderado  $G = \langle V; E \rangle$  con función de peso  $w : E \rightarrow \mathbb{R}$  y los vértices de origen tales que  $G$  satisface la siguiente propiedad: Para cada arista  $e = (u, v) \in E$ , hay un árbol de caminos más cortos enraizado en  $s$  que contiene  $e$ ; y otro árbol de caminos más cortos arraigado en  $s$  que no contiene  $e$ .

## 24.5-3

Adorne la prueba del Lema 24.10 para manejar los casos en los que los pesos del camino más corto son 1 o -1.

## 24.5-4

Sea  $G = \langle V; E \rangle$  sea un grafo dirigido y ponderado con vértice fuente  $s$ , y sea  $G$  inicializado por INITIALIZE-SINGLE-SOURCE( $G; s$ ). Demuestre que si una secuencia de pasos de relajación establece  $s$ : en un valor distinto de NIL, entonces  $G$  contiene un ciclo de peso negativo.

## 24.5-5

Sea  $G = \langle V; E \rangle$  ser un gráfico dirigido y ponderado sin bordes de peso negativo. Sea  $s \in V$  el vértice fuente, y supongamos que permitimos que : sea el predecesor de en cualquier camino más corto desde la fuente  $s$  si  $v$  es accesible desde  $s$ , y NIL en caso contrario. Dé un ejemplo de tal gráfico  $G$  y una asignación de valores que produzca un ciclo en  $G$ . (Por el Lema 24.16, tal asignación no puede ser producida por una secuencia de pasos de relajación).

## 24.5-6

Sea  $G = \langle V; E \rangle$  sea un gráfico dirigido y ponderado con una función de peso  $w : E \rightarrow \mathbb{R}$  y sin ciclos de peso negativo. Sea  $s \in V$  el vértice fuente, y sea  $G$  inicializado por INITIALIZE-SINGLE-SOURCE( $G; s$ ). Demostrar que para todo vértice  $v \in V$  existe un camino de  $s$  a  $v$  en  $G$  y que esta propiedad se mantiene invariante sobre cualquier secuencia de relajaciones.

## 24.5-7

Sea  $G = \langle V; E \rangle$  sea una gráfica dirigida y ponderada que no contenga ciclos de ponderación negativa. Sea  $s \in V$  el vértice fuente, y sea  $G$  inicializado por INITIALIZE SINGLE-SOURCE( $G; s$ ). Demuestre que existe una secuencia de  $j$  pasos de relajación que produce :d  $i.s$ ; / para todos los  $v \in V$ .

## 24.5-8

Sea  $G$  un grafo dirigido, ponderado arbitrariamente, con un ciclo de peso negativo accesible desde el vértice fuente  $s$ . Muestre cómo construir una secuencia infinita de relajaciones de los bordes de  $G$  tal que cada relajación provoque un cambio en la estimación del camino más corto.

## Problemas

**24-1 Mejora de Yen a Bellman-Ford** Suponga que ordenamos las relajaciones de los bordes en cada paso del algoritmo de Bellman-Ford de la siguiente manera. Antes del primer paso, asignamos un orden lineal arbitrario  $1; 2; \dots; j; V$  a los vértices del grafo de entrada  $G$ .  $V = M$ . Luego, dividimos el conjunto de aristas  $E$  en  $E_f$  [  $E_b$ , donde  $E_f$   $D_{fi}$ ;  $j / 2$   $E_W$   $i < j$  y  $E_b$   $D_{fi}$ ;  $j / 2$   $E_W$   $i > j$ . (Suponga que  $G$  no contiene bucles propios, de modo que cada arista está en  $E_f$  o  $E_b$ .) Defina  $G_f$   $D_{fi}$ ;  $E_f$  / y  $G_b$   $D_{bi}$ ;  $E_b$ .

- a. Demuestre que  $G_f$  es acíclico con orden topológico  $h_1; 2; \dots; j; V$  y que  $G_b$  es acíclico con orden topológico  $h_j; V; j; V; j_1; \dots; 1; i$ .

Suponga que implementamos cada paso del algoritmo Bellman-Ford de la siguiente manera. Visitamos cada vértice en el orden  $1; 2; \dots; j; V$ , relajando aristas de  $E_f$  que salen del vértice. Luego visitamos cada vértice en el orden  $j; V; j; V; j_1; \dots; 1$ , aristas relajadas de  $M_b$  que salen del vértice.

- b. Demuestre que con este esquema, si  $G$  no contiene ciclos de peso negativo que sean alcanzables desde el vértice fuente  $s$ , entonces después de que solo  $d_{jV} = \infty$  pase por los bordes,  $:d_D i.s ; /$  para todos los vértices  $2 \leq V$ .

C. ¿Este esquema mejora el tiempo de funcionamiento asintótico de Bellman-Ford?  
¿algoritmo?

**24-2 Cajas nido Una caja**

de dimensiones  $d$  con dimensiones  $x_1; x_2; \dots; x_d$  / anida dentro de otra caja de dimensiones  $y_1; y_2; \dots; y_d$  / si existe una permutación en  $f_1; 2; \dots; d$  tal que  $x_1 < y_1, x_2 < y_2, \dots, x_d < y_d$ .

- a. Argumente que la relación de anidamiento es transitiva.

b. Describa un método eficiente para determinar si un  $d$ -dimensional caja anida dentro de otra.

C. Suponga que le dan un conjunto de cajas  $B_1$  de  $n$   $d$  dimensiones ;  $B_2; \dots; B_n$ . Dé un algoritmo eficiente para encontrar la secuencia más larga  $hB_1; B_2; \dots; B_k$  de cajas tales que  $B_i$  anida dentro de  $B_j$  para  $j \leq 1; 2; \dots; k$ . Exprese el tiempo de ejecución de su algoritmo en términos de  $n$  y  $d$ .

### 24-3 Arbitraje El

arbitraje es el uso de discrepancias en los tipos de cambio de divisas para transformar una unidad de una divisa en más de una unidad de la misma divisa. Por ejemplo, suponga que 1 dólar estadounidense compra 49 rupias indias, 1 rupia india compra 2 yenes japoneses y 1 yen japonés compra 0:0107 dólares estadounidenses. Luego, al convertir monedas, un operador puede comenzar con 1 dólar estadounidense y comprar 4920:0107 D 1:0486 dólares estadounidenses, obteniendo así una ganancia de 4:86 por ciento.

Supongamos que nos dan n monedas  $c_1; c_2; \dots; c_n$  y una tabla nn R de tipos de cambio, tal que una unidad de moneda  $c_i$  compra  $R_{c_i}^c$ ; j unidades de moneda  $c_j$ .

- a. Dar un algoritmo eficiente para determinar si existe o no una secuencia

de monedas  $c_1; c_2; \dots; c_k$  i tal que

$$R_{c_1}^c; i_2 R_{c_2}^c; i_3 R_{c_3}^c; \dots; i_k R_{c_k}^c; i_1 > 1$$

Analice el tiempo de ejecución de su algoritmo.

- b. Proporcione un algoritmo eficiente para imprimir tal secuencia, si existe. Analice el tiempo de ejecución de su algoritmo.

### 24-4 Algoritmo de escalamiento de Gabow para rutas más cortas de fuente

única Un algoritmo de escalamiento resuelve un problema al considerar inicialmente solo el bit de mayor orden de cada valor de entrada relevante (como un peso de borde). Luego refina la solución inicial observando los dos bits de mayor orden. Progresivamente examina más y más bits de alto orden, refinando la solución cada vez, hasta que ha examinado todos los bits y calculado la solución correcta.

En este problema, examinamos un algoritmo para calcular las rutas más cortas desde una sola fuente escalando los pesos de los bordes. Nos dan un grafo dirigido  $G = (V, E)$  con pesos de borde de enteros no negativos  $w$ . Sea  $W = \max_{(u,v) \in E} w_{u,v}$ . Nuestro objetivo es desarrollar un algoritmo que se ejecute en  $O(E \lg W / \text{time})$ . Suponemos que todos los vértices son accesibles desde la fuente.

El algoritmo descubre los bits en la representación binaria de los pesos de borde de uno en uno, desde el bit más significativo hasta el bit menos significativo. Específicamente, sea  $k = \lceil \lg W \rceil$  el número de bits en la representación binaria de  $W$  y para  $i = 1; 2; \dots; k$ , vamos a definir  $w_i = w / 2^{k-i}$ . Es decir,  $w_i$  es la versión "reducida" de  $w$ .

(Así,  $w_k = w / 2^{k-k} = 1$  para todo  $i \leq k$ .) Por ejemplo, si  $w = 25$ , que tiene la representación binaria  $11001_2$ , luego  $w_1 = 1$ ,  $w_2 = 1$ ,  $w_3 = 1$ ,  $w_4 = 1$ ,  $w_5 = 1$ . Como otro ejemplo con  $w = 5$ , si  $w_1 = 1$ ,  $w_2 = 1$ ,  $w_3 = 1$ ,  $w_4 = 1$ ,  $w_5 = 1$ . Definamos  $\pi_i(u)$  como el peso de la ruta más corta desde el vértice  $u$  hasta el vértice usando la función de peso  $w_i$ . Por lo tanto,  $\pi_i(u) = \min_{v \in V} w_i(v, u)$  para todos ustedes; 2 voltios Para un vértice fuente dado, el algoritmo de escalado primero calcula el

pesos de ruta más corta  $i_1.s$ ; / para todos los  $V$ , luego calcula  $i_2.s$ ; / para todos  $V$ ,  $i_2.s$  y así sucesivamente, hasta que calcule  $i_k.s$ ; / para todos los  $V$ . Asumimos en todo momento que  $j \in V \setminus \{i_1\}$ , y veremos que calcular  $i_i$  a partir de  $i_1$  toma  $O(kE)$  en  $O(E \lg W)$  tiempo.

- a. Supongamos que para todos los vértices  $V$ , tenemos  $i.s$ ; / J.Ej. Demostrar que podemos calcular  $i.s$ ; / para todos los  $V$  en  $O(E \lg W)$  tiempo.

- b. Demuestre que podemos calcular  $i_1.s$ ; / para todos los  $V$  en  $O(E \lg W)$  tiempo.

Centrémonos ahora en calcular  $i_i$  a partir de  $i_1$ .

- C. Demostrar que para  $i \in D; 3; \dots; k$ , tenemos  $w_{i,u} / D 2w_{i_1,u} / o w_{i,u} / D 2w_{i_1,u} / C 1$ . Luego, demuestre que

$$2i_1.s; / i_i.s; / 2i_1.s; / C jV j \quad 1$$

para todos los  $V$ .

- d. Definir para  $i \in D; 3; \dots; k$  y todo  $u \in V$ ,  $w_{i,u} / D w_{i_1,u} / C 2i_1.s; u / 2i_1.s; / :$

Demostrar que para  $i \in D; 3; \dots; k$  y todo  $u \in V$ , el valor “reponderado”  $w_{i,u} /$  de arista  $u$  es un entero no negativo.

mi. Ahora, defina  $y_{i,s}$ ; / como el peso del camino más corto desde  $s$  hasta usar el peso función  $w_{i,s}$ . Demostrar que para  $i \in D; 3; \dots; k$  y todos  $V$ ,

$$i_i.s; / D i_i.s; / C 2i_1.s; /$$

y que  $y_{i,s} / jEj$ .

- F. Muestre cómo calcular  $i_i.s$ ; / de  $i_1.s$ ; / para todos los  $V$  en  $O(E \lg W)$  tiempo, y concluya que podemos calcular  $i.s$ ; / para todos los  $V$  en  $O(E \lg W)$  tiempo.

#### 24-5 Algoritmo del ciclo de peso medio mínimo de Karp

Sea  $G = (V, E)$  sea un grafo dirigido con función de peso  $w : E \rightarrow \mathbb{R}$ , y sea  $n \in V$ .

Definimos el peso medio de un ciclo  $c \in E$  como la media de pesos de las aristas en  $E$  ser

$$\frac{1}{|c|} \sum_{e \in c} w(e) :$$

Sea  $D_{minc}$  .c/, donde c varía sobre todos los ciclos dirigidos en G. Llamamos a un ciclo c para el cual .c/ D un ciclo de peso medio mínimo. Este problema investiga un algoritmo eficiente para la computación.

Suponga sin pérdida de generalidad que todo vértice 2 V es alcanzable desde a y sea fuente vértice s 2 V . Dejar que se; / ser el peso de un camino más corto de s , a ik.s; / ser el peso de un camino más corto de s a que consta exactamente de k aristas. Si no hay un camino desde s hasta con exactamente k aristas, entonces ik.s; / D 1.

- a. Muestre que si D 0, entonces G no contiene ciclos de peso negativo y s; / D min0kn1 ik.s; / para todos los vértices 2 V .

- b. Demuestre que si D 0, entonces

$$\frac{\text{En } s; / ik.s; /}{\substack{\text{máximo } 0kn1 \\ nk}} \quad 0$$

para todos los vértices 2 V . (Sugerencia: use ambas propiedades de la parte (a).)

- C. Sea c un ciclo de peso 0, y sean uy dos vértices cualesquiera de c. Suponga que D 0 y que el peso del camino simple de u a lo largo del ciclo es x. Demostrar que i.s; / D i.s; tu/C x. (Sugerencia: el peso del camino simple desde u a lo largo del ciclo es x).

- d. Muestre que si D 0, entonces en cada ciclo de peso medio mínimo existe un vértice tal que

$$\frac{\text{En } s; / ik.s; / nk}{\substack{\text{máximo } 0kn1}} \quad D 0$$

(Sugerencia: muestre cómo extender una ruta más corta a cualquier vértice en un ciclo de peso medio mínimo a lo largo del ciclo para hacer una ruta más corta al siguiente vértice en el ciclo).

- mi. Demuestre que si D 0, entonces

$$\min_{2V} \frac{\text{En } s; / ik.s; / nk}{\text{máximo } 0kn1} \quad D 0$$

- F. Demuestre que si sumamos una constante t al peso de cada arista de G, entonces aumenta en t. Utilice este hecho para demostrar que

$$D_{mín.} \quad \frac{\text{En } s; / ik.s; / nk}{\substack{2V \\ \text{máximo } 0kn1}}$$

gramo. Proporcione un algoritmo O.VE/-tiempo para calcular .

#### 24-6 Caminos más cortos bitónicos

Una secuencia es bitónica si aumenta monótonamente y luego decrece monótonamente, o si por un desplazamiento circular aumenta monótonamente y luego decrece monótonamente. Por ejemplo las secuencias  $h_1; 4; 6; 8; 3; 2i, h_9; 2; 4; 10; 5i$  y  $h_1; 2; 3; 4i$  son bitónicas, pero  $h_1; 3; 12; 4; 2; 10i$  no es bitónico. (Vea el Problema 15-3 para el problema del vendedor ambulante euclíadiano bitónico.)

Supongamos que nos dan un grafo dirigido  $G = V; E$  con función de peso  $w: V \times V \rightarrow \mathbb{R}$ , donde todos los pesos de los bordes son únicos, y deseamos encontrar las rutas más cortas de fuente única desde un vértice de fuente  $s$ . Se nos da una pieza adicional de información para los pesos para cada vértice  $v \in V$  de  $s$  para formar, de los bordes a lo largo de cualquier camino más corto: una secuencia bitónica.

Proporcione el algoritmo más eficiente que pueda para resolver este problema y analice su tiempo de ejecución.

#### Notas del capítulo

El algoritmo de Dijkstra [88] apareció en 1959, pero no contenía ninguna mención de una cola de prioridad. El algoritmo de Bellman-Ford se basa en algoritmos separados de Bellman [38] y Ford [109]. Bellman describe la relación de los caminos más cortos con las restricciones de diferencia. Lawler [224] describe el algoritmo de tiempo lineal para los caminos más cortos en un dag, que considera parte del folclore.

Cuando los pesos de los bordes son números enteros no negativos relativamente pequeños, tenemos algoritmos más eficientes para resolver el problema de los caminos más cortos de fuente única. La secuencia de valores devueltos por las llamadas EXTRACT-MIN en el algoritmo de Dijkstra aumenta monótonamente con el tiempo. Como se discutió en las notas del capítulo 6, en este caso, varias estructuras de datos pueden implementar las diversas operaciones de cola de prioridad de manera más eficiente que un montón binario o un montón de Fibonacci. Ahuja, Mehlhorn, Orlin y Tarjan [8] dan un algoritmo que se ejecuta en  $O(|E| \sqrt{|V|} \log |V|)$  tiempo en gráficos con pesos de borde no negativos, donde  $|W|$  es el mayor peso de cualquier borde en el gráfico. Los mejores límites son los de Thorup [337], que proporciona un algoritmo que se ejecuta en  $O(|E| \lg \lg |V| \log \lg |V|)$  tiempo, y de Raman [291], que proporciona un algoritmo que se ejecuta en  $O(|E| \lg |V| \lg \lg |V|)$  tiempo. Estos dos algoritmos usan una cantidad de espacio que depende del tamaño de palabra de la máquina subyacente. Aunque la cantidad de espacio utilizado puede ser ilimitado en el tamaño de la entrada, se puede reducir para que sea lineal en el tamaño de la entrada mediante hash aleatorio.

Para gráficos no dirigidos con pesos enteros, Thorup [336] da un algoritmo de tiempo  $O(|E| \lg |V| \lg \lg |V|)$  para caminos más cortos de fuente única. A diferencia de los algoritmos mencionados en el párrafo anterior, este algoritmo no es una implementación de Dijk-

algoritmo de stra, ya que la secuencia de valores devueltos por las llamadas EXTRACT-MIN no aumenta monótonamente con el tiempo.

Para gráficos con pesos de borde negativos, un algoritmo debido a Gabow y Tarjan [122] se ejecuta en  $O.pV E \lg.VW // \text{tiempo}$ , y uno de Goldberg [137] se ejecuta en  $O.pV E \lg W / \text{tiempo}$ , donde  $WD \max.u; 2E fju.u; /jg$ .

Cherkassky, Goldberg y Radzik [64] realizaron extensos experimentos com emparejando varios algoritmos de ruta más corta.

En este capítulo, consideramos el problema de encontrar los caminos más cortos entre todos los pares de vértices en un gráfico. Este problema podría surgir al hacer una tabla de distancias entre todos los pares de ciudades para un atlas de carreteras. Como en el capítulo 24, tenemos un gráfico dirigido ponderado  $G = \langle V, E \rangle$  con una función de peso  $w : E \rightarrow \mathbb{R}$  que asigna bordes a pesos de valor real. Deseamos encontrar, para cada par de vértices  $u, v \in V$ , camino más corto  $\pi_{uv}$  (menor peso) desde  $u$  hasta  $v$ ; donde el peso de  $\pi_{uv}$  es la suma de los pesos de sus aristas constituyentes. Por lo general, queremos la salida en forma tabular: la entrada en la fila de  $u$  y la columna de  $v$  debe ser el peso de la ruta más corta de  $u$  a  $v$ .

Podemos resolver un problema de caminos más cortos de todos los pares ejecutando un algoritmo de caminos más cortos de fuente única  $j$  veces, una vez para cada vértice como fuente. Si todos los pesos de los bordes son no negativos, podemos usar el algoritmo de Dijkstra. Si usamos la implementación de matriz lineal de la cola de prioridad mínima, el tiempo de

<sup>3</sup> ejecución es  $O(|V|^2 |E|)$ . La implementación binaria min-heap de la cola OV de prioridad mínima produce un tiempo de ejecución de  $O(|V| \log |V|)$ , lo que es una mejora si el gráfico es escaso. Alternativamente, podemos implementar la cola de prioridad mínima con un montón de Fibonacci, lo que produce un tiempo de ejecución de  $O(|V|^2 \log |V|)$ .

Si el gráfico tiene bordes de peso negativo, no podemos usar el algoritmo de Dijkstra. En su lugar, debemos ejecutar el algoritmo más lento de Bellman-Ford una vez desde cada vértice. El tiempo de ejecución resultante es  $O(|V|^2 |E|)$ , que en un gráfico denso es  $O(|V|^3)$ . En este capítulo veremos cómo hacerlo mejor. También investigamos la relación del problema de los caminos más cortos de todos los pares con la multiplicación de matrices y estudiamos su estructura algebraica.

A diferencia de los algoritmos de fuente única, que asumen una representación de lista de adyacencia del gráfico, la mayoría de los algoritmos de este capítulo utilizan una representación de matriz de adyacencia. (El algoritmo de Johnson para grafos dispersos, en la Sección 25.3, usa listas de adyacencia.) Por conveniencia, asumimos que los vértices están numerados  $1, 2, \dots, j$ , de modo que la entrada es una matriz  $W$  de  $|V| \times |V|$  que representa los pesos de los bordes de un gráfico  $G = \langle V, E \rangle$  dirigido por  $n$  vértices;  $W[i][j]$  es decir,  $w_{ij}$ , donde

	0	si yo D j ;	
wij d	el peso del borde dirigido .i; j / si i ≠ j y .i; j / 2 E si i ≠ j y .i; j / 62 E		(25.1)
	1		

Permitimos bordes de peso negativo, pero asumimos por el momento que el gráfico de entrada no contiene ciclos de peso negativo.

La salida tabular de los algoritmos de caminos más cortos de todos los pares presentados en este capítulo es una matriz  $D$  donde la entrada  $D_{ij}$  contiene el peso de un camino más corto desde el vértice  $i$  al vértice  $j$ , es decir, si dejamos  $D_{ij}$  denota el peso del camino más corto desde el vértice  $i$  al vértice  $j$  (como en el Capítulo 24), luego  $D_{ij} = \infty$  en terminación.

Para resolver el problema de los caminos más cortos de todos los pares en una matriz de adyacencia de entrada, necesitamos calcular no solo los pesos del camino más corto sino también una matriz de predecesora ...  $D_{ij}$ , donde es NIL si  $i = j$  o no hay camino de  $i$  a  $j$  de otra manera  $D_{ij}$  es el predecesor de  $j$  en algún camino más corto desde  $i$ . Así como el subgrafo predecesor  $G$  del Capítulo 24 es un árbol de caminos más cortos para un vértice fuente dado, el subgrafo inducido por la  $i$ -ésima fila de la matriz ... debería ser un árbol de caminos más cortos con raíz  $i$ . Para cada vértice  $i$  definimos el subgrafo predecesor de  $G$  para  $i$  como  $G_i = \{v_j \mid D_{ij} < \infty\}$ , donde

$V_i = \{j \mid D_{ij} < \infty\}$   $D_{ij} = \text{NIL}$  si  $j \notin V_i$

y

$E_i = \{(i, j) \mid D_{ij} < \infty\}$

Si  $G_i$  es un árbol de rutas más cortas, entonces el siguiente procedimiento, que es una versión modificada del procedimiento PRINT-PATH del Capítulo 22, imprime una ruta más corta desde el vértice  $i$  hasta el vértice  $j$ .

```

IMPRIMIR-TODOS-LOS-PARES-RUTA-MAS-CORTA....;
i; j / 1 si i == j
2 imprimir i 3 otra
cosa      yo == NINGUNO
          print "ninguna ruta desde" i "a" j "existe" 4 5
else IMPRIMIR-TODOS-LOS-PARES-RUTA-MAS-CORTA....;    yo /
i; 6 imprimir j

```

Para resaltar las características esenciales de los algoritmos de todos los pares en este capítulo, no cubriremos la creación y las propiedades de las matrices predecesoras tan extensamente como lo hicimos con los subgrafos predecesores en el Capítulo 24. Algunos de los ejercicios cubren los conceptos básicos.

### Bosquejo del capítulo

La sección 25.1 presenta un algoritmo de programación dinámica basado en la multiplicación de matrices para resolver el problema de los caminos más cortos de todos los pares. Usando la técnica de “cuadrado repetido”, podemos lograr un tiempo de  $V^3 \lg V$ . La sección 25.2 da ejecución de .V otro algoritmo de programación dinámica, el algoritmo de Floyd-Warshall, que se ejecuta en el tiempo  $V^3$ . La sección 25.2 también trata el problema de encontrar la clausura transitiva de un grafo dirigido, que está relacionado con el problema de los caminos más cortos de todos los pares. Finalmente, la Sección 25.3 presenta el algoritmo de Johnson, que resuelve el problema de los caminos más cortos de  $V^2 \lg VC VE$  tiempo y es una buena opción para todos los pares en grafos grandes y dispersos de OV.

Antes de continuar, necesitamos establecer algunas convenciones para las representaciones de matriz de adyacencia. Primero, supondremos generalmente que el gráfico de entrada GD .V; E/ tiene n vértices, de modo que n D jV j. En segundo lugar, usaremos la convención de denotar matrices con letras mayúsculas, como W, L o D, y sus elementos individuales con letras minúsculas subíndice, como  $w_{ij}$ ,  $l_{ij}$  o  $d_{ij}$ . Algunas matrices tendrán o  $D_m$  / D  $d_m$  / superíndices entre paréntesis, Finalmente, para una matriz A dada de nn, supondremos  $A^{m/}$  como en  $L_m$  / D  $I_m$  iterar. , indicar que el valor de n está almacenado en el atributo A:rows.

## 25.1 Caminos más cortos y multiplicación de matrices

Esta sección presenta un algoritmo de programación dinámica para el problema de los caminos más cortos de todos los pares en un grafo dirigido GD .V; MI/. Cada ciclo principal del programa dinámico invocará una operación que es muy similar a la multiplicación de matrices, de modo que el algoritmo parecerá una multiplicación de matrices repetida. Comenzaremos desarrollando un algoritmo .V 4-time para el problema de los caminos más cortos de todos los pares y luego mejoraremos su tiempo de ejecución a .V  $V^3 \lg V$ .

Antes de continuar, recapitulemos brevemente los pasos dados en el capítulo 15 para desarrollar un algoritmo de programación dinámica.

1. Caracterizar la estructura de una solución óptima.
2. Definir recursivamente el valor de una solución óptima.
3. Calcular el valor de una solución óptima de forma ascendente.

Reservamos el cuarto paso, construir una solución óptima a partir de información calculada, para los ejercicios.

La estructura de un camino más corto.

Comenzamos por caracterizar la estructura de una solución óptima. Para el problema de los caminos más cortos de todos los pares en un gráfico  $G = (V, E)$ , hemos demostrado (Lema 24.1) que todos los subcaminos de un camino más corto son caminos más cortos.

Supongamos que representamos el grafo mediante una matriz de adyacencia  $W = (w_{ij})$ .

Considere un camino más corto  $p$  desde el vértice  $i$  al vértice  $j$ , y suponga que  $p$  contiene como máximo  $m$  aristas. Suponiendo que no hay ciclos de peso negativo,  $m$  es finito. Si  $i \neq j$ , entonces  $p$  tiene peso 0 y no tiene aristas. Si los vértices  $i$  y  $j$  son distintos, entonces

descomponemos el camino  $p$  en  $p_0 \dots k \dots j$ , donde la ruta  $p_0$  ahora contiene como máximo  $m-1$  aristas. Por el Lema 24.1,  $p_0$  es el camino más corto de  $i$  a  $k$ , y así  $i \rightarrow j \geq i \rightarrow k \geq C(w_{kj})$ .

Una solución recursiva al problema de los caminos más cortos de todos los pares

.m/ Ahora, sea  $I$  el peso mínimo de cualquier camino desde el vértice  $i$  al vértice  $j$  que  $i \neq j$  contiene como máximo  $m$  aristas. Cuando  $m=0$ , hay un camino más corto de  $i$  a  $j$  sin bordes si y solo si  $i = j$ . De este modo,

$$\begin{aligned} I_{ij}^{(0)} &= \text{si } i = j \\ D &= \begin{cases} 0 & \text{si } i \neq j \end{cases} \end{aligned}$$

Para  $m=1$ , calculamos  $I$  (el peso <sup>metro</sup><sub>yo</sub> de un camino más corto de  $i$  a  $j$  que consta de como máximo  $m=1$  aristas) y el peso mínimo de cualquier camino de  $i$  a  $j$  que consta de como máximo  $m$  aristas, obtenido al observar todos los posibles predecesores  $k$  de  $j$ . Por lo tanto, definimos recursivamente

$$\begin{aligned} I_{ij}^{(m)} &= \min I_{ik} + \min_{1 \leq k \leq n} I_{kj}^{(m-1)} \\ D &= \min_{\text{nodo } i} \max_{1 \leq k \leq n} I_{kj}^{(m-1)} \end{aligned} \quad (25.2)$$

La última igualdad sigue ya que  $w_{ij} \geq 0$  para todo  $j$ .

¿Cuáles son los pesos reales del camino más corto  $i \rightarrow j$ ? Si el gráfico no contiene ciclos de peso negativo, entonces para cada par de vértices  $i, j$  para los cuales  $i \rightarrow j \leq 1$ , hay un camino más corto de  $i$  a  $j$  que es simple y, por lo tanto, contiene como máximo  $n-1$  aristas. Un camino del vértice  $i$  al vértice  $j$  con más de  $n-1$  aristas no puede tener un peso menor que el camino más corto de  $i$  a  $j$ . Por lo tanto, los pesos reales del camino más corto están dados por

$$\min_{1 \leq k \leq n} w_{ik} \quad (25.3)$$

Cálculo de los pesos de la ruta más corta de abajo hacia arriba

Tomando como entrada la matriz  $WD .w_{ij} /$ , ahora calculamos una serie de matrices  $L.1/; L.2/; \dots; L.n1/$ , donde para  $m D 1; 2; \dots; n 1$ , tenemos  $Lm/ D I_{yo}^{.metro/}$ . La matriz final  $L.n1/$  contiene los pesos reales del camino más corto. Obsérvese que  $L.m1/ y así L.1/ DW .w_{ij} /$ , para todos los vértices  $i; j 2 V$ ,

El corazón del algoritmo es el siguiente procedimiento, que, dadas las matrices,  $L.m1/ y W$ , devuelve la matriz  $Lm/$ . Es decir, extiende los caminos más cortos calculados hasta ahora por un borde más.

```

EXTENDER-RUTAS MÁS CORTAS.L; W /
1 n D L.filas 2 sea
L0 D I0 ij 3 para i D ser una nueva matriz nn
1 a n 4
      para j D 1 a n
5      I0 D 1
      ij para k D 1 a
      I0 ij n D min.I0 ; como C wkj /
6 7 8 volver L0

```

El procedimiento calcula una matriz  $L0_{yo}/$ , que devuelve al final. lo hace  $D .I0$  calculando la ecuación (25.2) para todo  $i y j$ , usando  $L$  para  $L.m1/ y L0$  para  $Lm/$ . (Está escrito sin los superíndices para que sus matrices de entrada y salida sean independientes de  $m$ ). Su tiempo de ejecución es  $,n^3$  debido a los tres bucles for anidados.

Ahora podemos ver la relación con la multiplicación de matrices. Supongamos que deseamos calcular el producto matricial  $CDAB$  de dos matrices  $A$  y  $B$  de  $nn$ . Entonces, para  $i;j D 1; 2; \dots; n$ , calculamos

$$c_{ij} D X_n \quad a_{ik} b_{kj} : \quad (25.4)$$

$$\quad \quad \quad kD1$$

Observa que si hacemos las sustituciones

$$\begin{aligned}
I .m1/ &= ! a & ; \\
w! b! .m/ &= ; \\
! C &= ; \\
\min! C &= ; \\
! C! &
\end{aligned}$$

en la ecuación (25.2), obtenemos la ecuación (25.4). Por lo tanto, si hacemos estos cambios en EXTEND-SHORTEST-PATHS y también reemplazamos 1 (la identidad de min) por 0 (la

identidad para C), obtenemos el mismo procedimiento „n3/-tiempo para multiplicar matrices cuadradas que vimos en la Sección 4.2:

CUADRADO-MATRIZ-MULTIPLICAR.A;

B/ 1 n D A:filas 2

sea C una nueva matriz nn 3 para i

D 1 a n 4 5 6 7 8

para j D 1 a n cij

D 0 para

k D 1 a n cij D cij

C aik bkj

devuelve C

Volviendo al problema de los caminos más cortos de todos los pares, calculamos los pesos de los caminos más cortos extendiendo los caminos más cortos borde por borde. Si AB denota el “producto” de la matriz devuelto por EXTEND-SHORTEST-PATHS.A; B/, calculamos la secuencia de n 1 matrices

L.1/ D L.0/ WDW L.2 / D L.1/ WDW ;

L.3 / D L.2/ WDW ;

3 ;

⋮

L.n1/ D L.n2/ WD W ;

n1 ;

Como argumentamos anteriormente, la matriz L.n1/ DW contiene los pesos de ruta más corta. El siguiente procedimiento calcula esta secuencia en „n4/ tiempo.

LENTO-TODOS-LOS-PARES-CAMINOS-MÁS-CORTOS.W /

1 n D W:rows 2

L.1/ DW 3 para

m D 2 a n 1 sea Lm/ una

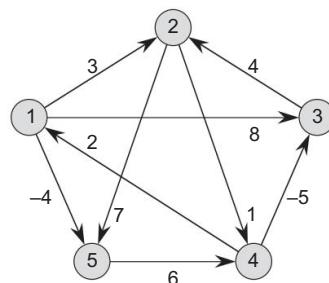
nueva matriz nn 4 Lm/ D EXTEND-

SHORTEST-PATHS.L.m1/;W/ 5 6 return L. n1/

La figura 25.1 muestra un gráfico y las matrices Lm/ calculadas por el procedimiento LENTO-TODOS-LOS-PARES-CAMINOS-MÁS-CORTOS.

Mejorar el tiempo de ejecución

Nuestro objetivo, sin embargo, no es calcular todas las matrices Lm/ : solo nos interesa la matriz L.n1/. Recuerde que en ausencia de ciclos de peso negativo, la ecuación



$$\begin{array}{c}
 \begin{array}{ccccc}
 0 & 3 & 8 & 1 & 4 \\
 1 & 0 & 1 & 1 & \\
 L.1/D & 1 & 4 & & 0 & 1 & 1 \\
 & 2 & 1 & 5 & 0 & 1 \\
 & 1 & 1 & 6 & & \\
 \end{array} & 
 \begin{array}{c}
 7 \\
 0^{\sim} \\
 \end{array} & 
 \begin{array}{ccccc}
 0 & 3 & 8 & 2 & 4 \\
 3 & 0 & 4 & 1 & 7 \\
 L.2/D & 1 & 4 & & 0 & 5 & 1 & 1 \\
 & 2 & 1 & 5 & 0 & & 2 \\
 & 8 & 1 & 1 & 6 & 0 & ^{\sim} \\
 \end{array} \\
 \\[10pt]
 \begin{array}{ccccc}
 0 & 3 & 3 & 2 & 4 \\
 3 & 0 & 4 & 1 & 1 \\
 L.3/D & 7 & 4 & 0 & 5 & 1 & 1 \\
 & 2 & 1 & 5 & 0 & 2 \\
 & 8 & 5 & 1 & 6 & 0 & ^{\sim} \\
 \end{array} & 
 \begin{array}{c}
 L.4/D \\
 0^{\sim} \\
 \end{array} & 
 \begin{array}{ccccc}
 0 & 1 & 3 & 2 & 4 \\
 3 & 0 & 4 & 1 & 1 \\
 7 & 4 & 0 & 5 & 3 \\
 2 & 1 & 5 & 0 & 2 \\
 8 & 5 & 1 & 6 & 0 & ^{\sim} \\
 \end{array}
 \end{array}$$

Figura 25.1 Un gráfico dirigido y la secuencia de matrices  $L_m$ / calculada por LENTO-TODOS-PARES-CAMINOS-MÁS-CORTOS. Es posible que desee verificar que  $L.5/$ , definido como  $L.4/ W$  , es igual a  $L.4/$  y, por lo tanto,  $L_m/ D L.4/$  para todo  $m \geq 4$ .

La ecuación (25.3) implica  $L_m/ D L.n1/$  para todos los enteros  $m \geq n \geq 1$ . Así como la multiplicación de matrices tradicional es asociativa, también lo es la multiplicación de matrices definida por el procedimiento EXTEND-SHORTEST-PATHS (vea el ejercicio 25.1-4). Por lo tanto, podemos calcular  $L.n1/$  con solo  $\text{dlg. } n \geq 1/e$  productos matriciales calculando el siguiente

$$\begin{array}{ccccccc}
 L.1/DW & & & & & & : \\
 L.2/DW L.4/DW & 2 & DW & W & & & : \\
 L.8/DW & 4 & DW & 2 & W & 2 & \\
 & 8 & DW & 4 & W & 4 & : \\
 \\[10pt]
 L.2\text{dlg. } n1/e/DW & 2\text{dlg. } n1/e & DW & 2\text{dlg. } n1/e & W & 2\text{dlg. } n1/e & :
 \end{array}$$

Como  $2\text{dlg. } n1/e \geq n \geq 1$ , el producto final  $L.2\text{dlg. } n1/e/$  es igual a  $L.n1/$ .

El siguiente procedimiento calcula la secuencia anterior de matrices usando este técnica del cuadrado repetido.

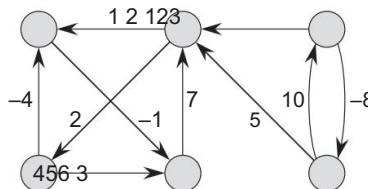


Figura 25.2 Gráfica dirigida ponderada para usar en los ejercicios 25.1-1, 25.2-1 y 25.3-1.

**MÁS RÁPIDO-TODOS-LOS-PARES-RUTAS**

MÁS CORTAS.W /

1 n D W:filas 2

L.1/ DW 3

m D 1 4 mientras que

m&lt; n 1 sea L.2m/ una nueva matriz

nn 5 L.2m/ D AMPLIAR-RUTAS MÁS CORTAS.Lm/; Im// 6

7 m P 2m 8 retorno

Lm/

En cada iteración del ciclo while de las líneas 4–7, calculamos  $L.2m/ D Lm/2$  comenzando con  $m = 1$ . Al final de cada iteración, duplicamos el valor de  $m$ . La iteración final calcula  $L.n1/$  calculando  $L.2m/$  para algún  $n \leq 2m < 2n$ . Por la ecuación (25.3),  $L.2m/ D L.n1/$ . La próxima vez que se realiza la prueba en la línea 4,  $m$  se ha duplicado, por lo que ahora es 1, la prueba falla y el procedimiento devuelve la última matriz que calculó.

Debido a que cada uno de los productos de la matriz  $Dlg.n 1/e$  toma  $.n^3/$  tiempo, FASTER ALL-PAIRS-SHORTEST-PATHS se ejecuta en  $.n^3 \lg n/$  tiempo. Observe que el código es estricto, no contiene estructuras de datos elaboradas y, por lo tanto, la constante oculta en la notación  $,\Theta$  es pequeña.

**Ejercicios****25.1-1**

Ejecute SLOW-ALL-PAIRS-SHORTEST-PATHS en el gráfico dirigido ponderado de la figura 25.2, que muestra las matrices que resultan para cada iteración del bucle. Luego haga lo mismo para FASTER-ALL-PAIRS-SHORTEST-PATHS.

**25.1-2**

¿Por qué requerimos que  $w_{ij} \geq 0$  para todo  $i, j \in V$ ?

## 25.1-3

¿Qué significa la matriz?

$$\begin{array}{c}
 \begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ L.0 / D & 1 & 1 & 0 \end{matrix} \\
 \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
 1 & 1 & 1 & 0
 \end{array}$$

utilizados en los algoritmos de caminos más cortos corresponden a la multiplicación regular de matrices?

## 25.1-4

Muestre que la multiplicación de matrices definida por EXTEND-SHORTEST-PATHS es asociativa.

## 25.1-5

Muestre cómo expresar el problema de los caminos más cortos de fuente única como un producto de matrices y un vector. Describa cómo la evaluación de este producto corresponde a un algoritmo tipo Bellman Ford (vea la Sección 24.1).

## 25.1-6

Suponga que también deseamos calcular los vértices en los caminos más cortos en los algoritmos de esta sección. Muestre cómo calcular la matriz predecesora... a partir de la matriz completa L de los pesos del camino más corto en  $O.n^3$ / tiempo.

## 25.1-7

También podemos calcular los vértices en los caminos más cortos como calculamos los pesos de los caminos más cortos.<sup>m/</sup> Defínase como el predecesor del vértice  $j$  en cualquier ruta de peso mínimo de  $i$  a  $j$  que contenga como máximo  $m$  aristas. Modificar los procedimientos EXTENDER-TRAYECTOS MÁS CORTOS y LENTO-TODOS-LOS-PARES-TRAYECTOS-MÁS-CORTOS para calcular las matrices  $\dots 1/; \dots 2/; \dots n/$  como las matrices  $L.1/; L.2/; \dots; L.n/$  se calculan.

## 25.1-8

El procedimiento MÁS RÁPIDO PARA TODOS LOS PARES DE LAS RUTAS MÁS CORTAS , tal como está escrito, requiere que almacenemos matrices  $dg.n$  1/e, cada una con  $n^2$  elementos, para un requisito de espacio total de  $.n^2 \lg n$ . Modifique el procedimiento para requerir solo el espacio  $.n^2$  usando solo dos matrices.

## 25.1-9

Modifique FASTER-ALL-PAIRS-SHORTEST-PATHS para que pueda determinar si el gráfico contiene un ciclo de peso negativo.

## 25.1-10

Proporcione un algoritmo eficiente para encontrar la longitud (número de aristas) de un ciclo de peso negativo de longitud mínima en una gráfica.

## 25.2 El algoritmo de Floyd-Warshall

En esta sección, usaremos una formulación de programación dinámica diferente para resolver el problema de los caminos más cortos de todos los pares en un grafo dirigido  $G = \langle V, E \rangle$ . El algoritmo resultante, conocido como algoritmo de Floyd-Warshall, se ejecuta en un tiempo de  $V^3$ . Como antes, los bordes de peso negativo pueden estar presentes, pero asumimos que no hay ciclos de peso negativo. Como en la Sección 25.1, seguimos el proceso de programación dinámica para desarrollar el algoritmo. Después de estudiar el algoritmo resultante, presentamos un método similar para encontrar el cierre transitivo de un gráfico dirigido.

La estructura de un camino más corto.

En el algoritmo de Floyd-Warshall, caracterizamos la estructura de un camino más corto de manera diferente a como la caracterizamos en la Sección 25.1. El algoritmo de Floyd-Warshall considera los vértices intermedios de un camino más corto, donde un vértice intermedio de un camino simple  $p$  es  $h_1; 2; \dots; l$  si es cualquier vértice de  $p$  distinto de 1 o  $l$ , es decir, cualquier vértice del conjunto  $f_2; 3; \dots; l$ .

El algoritmo de Floyd-Warshall se basa en la siguiente observación. Bajo nuestra suposición de que los vértices de  $G$  son  $V = f_1; 2; \dots; n$ , consideremos un subconjunto  $f_1; 2; \dots; k$  de vértices para algún  $k$ . Para cualquier par de vértices  $i, j \in V$ , considere todos los caminos de  $i$  a  $j$  cuyos vértices intermedios se extraen todos de  $f_1; 2; \dots; k$ , y sea  $p$  un camino de peso mínimo entre ellos. (La ruta  $p$  es simple). El algoritmo de Floyd Warshall explota una relación entre la ruta  $p$  y las rutas más cortas de  $i$  a  $j$  con todos los vértices intermedios en el conjunto  $f_1; 2; \dots; k$ . La relación depende de si  $k$  es o no un vértice intermedio del camino  $p$ .

Si  $k$  no es un vértice intermedio del camino  $p$ , entonces todos los vértices intermedios del camino  $p$  están en el conjunto  $f_1; 2; \dots; k$ . Así, un camino más corto del vértice  $i$  al vértice  $j$  con todos los vértices intermedios en el conjunto  $f_1; 2; \dots; k$  es también un camino más corto de  $i$  a  $j$  con todos los vértices intermedios en el conjunto  $f_1; 2; \dots; k$ .

Si  $k$  es un vértice intermedio del camino  $p$ , entonces descomponemos  $p$  en  $i \xrightarrow{k} p_1 \xrightarrow{j} p_2$  como ilustra la figura 25.3. Por el Lema 24.1,  $p_1$  es el camino más corto desde  $i$  hasta  $k$  con todos los vértices intermedios en el conjunto  $f_1; 2; \dots; k$ . De hecho, podemos hacer una declaración un poco más fuerte. Como el vértice  $k$  no es un vértice intermedio del camino  $p_1$ , todos los vértices intermedios de  $p_1$  están en el conjunto  $f_1; 2; \dots; k$ . Allá-

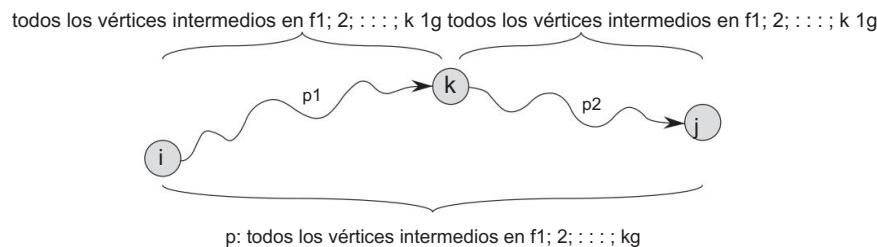


Figura 25.3 El camino  $p$  es el camino más corto desde el vértice  $i$  al vértice  $j$ , y  $k$  es el vértice intermedio con el número más alto de  $p$ . El camino  $p_1$ , la porción del camino  $p$  del vértice  $i$  al vértice  $k$ , tiene todos los vértices intermedios en el conjunto  $f_1; 2; \dots; k; 1g$ . Lo mismo vale para el camino  $p_2$  del vértice  $k$  al vértice  $j$ .

por tanto,  $p_1$  es un camino más corto de  $i$  a  $k$  con todos los vértices intermedios en el conjunto  $f_1; 2; \dots; k; 1g$ . De manera similar,  $p_2$  es el camino más corto desde el vértice  $k$  al vértice  $j$  con todos los vértices intermedios en el conjunto  $f_1; 2; \dots; k; 1g$ .

Una solución recursiva al problema de los caminos más cortos de todos los pares

Con base en las observaciones anteriores, definimos una formulación recursiva de estimaciones de la ruta más corta que difiere de la de la Sección 25.1. Sea  $d_{kj}$  el peso de un camino más corto desde el vértice  $i$  al vértice  $j$  para el cual todos los vértices intermedios están en el conjunto  $f_1; 2; \dots; k; g$ . Cuando  $k = 0$ , un camino desde el vértice  $i$  al vértice  $j$  sin vértice intermedio numerado más alto que 0 no tiene ningún vértice intermedio.

Tal camino tiene como máximo un borde, y por lo tanto  $d_{0j} = d_{ij}$ . Siguiendo la discusión anterior, definimos  $d_{kj}$  recursivamente por

$$d_{kj} = \min_{y_0} \left( \frac{d_{k1}}{y_0} + \dots + \frac{d_{kj}}{y_0} \right) \quad \begin{array}{ll} \text{si } k = 0 & ; \\ \text{si } k > 0 & ; \end{array} \quad (25.5)$$

Porque para cualquier camino, todos los vértices intermedios están en el conjunto  $f_1; 2; \dots; n$ , la matriz  $D_n$  da la respuesta final:  $d_{ij} = d_{i,j}$  para todo  $i, j \in V$ .

Cálculo de los pesos de la ruta más corta de abajo hacia arriba

Con base en la recurrencia (25.5), podemos usar el siguiente procedimiento de abajo hacia arriba para calcular los valores  $d_{kj}$  en orden de valores crecientes de  $k$ . Su entrada es una matriz  $W$  definida como en la ecuación (25.1). El procedimiento devuelve la matriz  $D_n$  de los pesos del camino más corto.

```

FLOYD-WARSHALL.W / 1
n D W: filas 2 D.0/
DW 3 para k D
1 a n
4      vamos Dk/ D dk/    ser una nueva matriz nn
        ij
        para i D 1 a n
5 6      para j D 1 a n
        dk/ D min d.k1/   yo : d.k1/ C d.k1/   kj
        yo          yo
7 8 volver Dn/

```

La figura 25.4 muestra las matrices  $D_k$  calculadas por el algoritmo de Floyd-Warshall para el gráfico de la figura 25.1.

El tiempo de ejecución del algoritmo de Floyd-Warshall está determinado por los bucles triplemente anidados de las líneas 3 a 7. Debido a que cada ejecución de la línea 7 toma el tiempo  $O(1)$ , el algoritmo se ejecuta en el tiempo  $.n^3$ . Como en el algoritmo final de la Sección 25.1, el código es estricto, sin estructuras de datos elaboradas, por lo que la constante oculta en la notación , es pequeña. Por lo tanto, el algoritmo de Floyd-Warshall es bastante práctico incluso para gráficos de entrada de tamaño moderado.

#### Construyendo un camino más corto

Hay una variedad de métodos diferentes para construir rutas más cortas en el algoritmo de Floyd Warshall. Una forma es calcular la matriz  $D$  de los pesos del camino más corto y luego construir la matriz predecesora... a partir de la matriz  $D$ . El ejercicio 25.1-6 le pide que implemente este método para que se ejecute en el tiempo  $O(n^3)$ . Dada la matriz anterior..., el procedimiento IMPRIMIR-TODOS-LOS-PARES-RUTA-MAS-CORTA imprimirá los vértices en una ruta más corta dada.

Alternativamente, podemos calcular la matriz predecesora... mientras que el algoritmo calcula las matrices  $D_k$ . Específicamente, calculamos una secuencia de matrices  $.k/ ....0/; ....1/; ...; ....n/$ , donde ...  $D ....n/$  y definimos como el predecesor de  $ij$  el vértice  $j$  en el camino más corto desde el vértice  $i$  con todos los vértices intermedios en el conjunto  $f1; 2; : : : ; kg$ .

Podemos dar una formulación recursiva de  $D_{ij}^{.k/}$ . Cuando  $k = 0$ , el camino más corto desde  $i$  no tiene vértices intermedios en absoluto. De este modo,

$$D_{ij}^{.0/} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases} \quad (25.6)$$

Para  $k = 1$ , si tomamos el camino  $i - k - j$ , donde  $k \neq j$ , entonces el predecesor de  $j$  que elegimos es el mismo que el predecesor de  $j$  que elegimos en el camino más corto desde  $k$  con todos los vértices intermedios en el conjunto  $f1; 2; : : : ; k$ . De lo contrario, nosotros

D.0/ D	03 8 1 4 1 0 1 1 1 4 0 1 1 2 1 5 0 1 1 1 1 6	7	NIL 1 1 NINGUNO 1 NIL NIL NIL 2 <b>NIL NIL NIL 5 NIL ^</b>
D.1/ D	03 8 1 4 1 0 1 1 1 4 0 1 1 2 5 5 0 1 1 1 6	7 2	NIL 1 1 NINGUNO 1 NIL NIL NIL 2 <b>NIL NIL NIL 5 NIL ^</b>
D.2/ D	0 3 8 4 4 1 0 1 1 1 4 0 5 1 1 2 5 5 0 1 1 6	7 2	NULO 1121 NIL NIL NIL 2 NINGUNO 3 NINGUNO 2 2 <b>NIL NIL NIL 5 NIL ^</b>
D.3/ D	0 3 8 4 4 1 0 1 1 1 4 0 5 1 1 2 1 5 0 1 1 1 6	7 2	NULO 1121 NIL NIL NIL 2 NINGUNO 3 NINGUNO 2 2 <b>NIL NIL NIL 5 NIL ^</b>
D.4/ D	0 3 1 4 4 3 0 4 1 1 7 4 0 5 3 ....4/ D 2 1 5 0 2 8 5 1 6 0 ^		NIL 1421 4 NULO 421 4 3 NINGUNO 2 1 434 NIL 1 <b>4345 NIL ^</b>
D.5/ D	0 1 3 2 4 0 4 1 1 3 7 4 0 5 3 ....5/ D 2 1 5 0 2 8 5 1 6 0 ^		NULO 3451 4 NULO 421 4 3 NINGUNO 2 1 434 NIL 1 <b>4345 NIL ^</b>

Figura 25.4 La secuencia de matrices  $D_k$  y  $\dots k$  calculada por el algoritmo de Floyd-Warshall para el gráfico de la Figura 25.1.

elegir el mismo predecesor de  $j$  que elegimos en un camino más corto desde  $i$  con todos los vértices intermedios en el conjunto  $f1; 2; \dots; k-1g$ . Formalmente, para  $k \geq 1$ ,

$$D_{ij}^{(k)} = \begin{cases} D_{ij}^{(k-1)} & \text{si } d_{kj} > d_{ij} \\ \min(d_{ij}, d_{ij} + d_{kj}) & \text{si } d_{kj} \leq d_{ij} \end{cases} \quad (25.7)$$

Dejamos la incorporación de los cálculos de la matriz  $\dots D_{ij}^{(k)}$  en el procedimiento de FLOYD WARSHALL como Ejercicio 25.2-3. La figura 25.4 muestra la secuencia de matrices  $\dots D_{ij}^{(k)}$  que el algoritmo resultante calcula para la gráfica de la figura 25.1. El ejercicio también pide la tarea más difícil de probar que el subgrafo predecesor  $G_i$  es un árbol de caminos más cortos con raíz  $i$ . El ejercicio 25.2-7 pide otra forma de reconstruir los caminos más cortos.

#### Cierre transitivo de un gráfico dirigido

Dado un grafo dirigido  $GD = \langle V, E \rangle$  con conjunto de vértices  $VD = f1; 2; \dots; ng$ , podríamos desear determinar si  $G$  contiene un camino de  $i$  a  $j$  para todos los pares de vértices  $i, j \in V$ . Definimos la clausura transitiva de  $G$  como el grafo  $GD' = \langle V, E' \rangle$ , donde

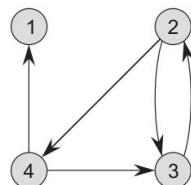
$E'D_{ij}$  si  $i, j \in V$  hay un camino del vértice  $i$  al vértice  $j$  en  $G$ :

Una forma de calcular el cierre transitivo de un gráfico en  $\sim n^3$  tiempo es asignar un peso de 1 a cada borde de  $E$  y ejecutar el algoritmo de Floyd-Warshall. Si hay un camino de  $i$  a  $j$ , obtenemos  $d_{ij} < n$ . De lo contrario, obtenemos

vértice  $j$ . Hay otra forma similar de calcular el cierre transitivo de  $G$  en  $\sim n^3$  tiempo que puede ahorrar tiempo y espacio en la práctica. Este método sustituye las operaciones lógicas  $\vee$  (OR lógico) y  $\wedge$  (AND lógico) por las operaciones aritméticas  $\min$  y  $+$  en el algoritmo de Floyd-Warshall. para  $y_0; j; k \in D = \{1, 2, \dots, n\}$ , definimos  $t_{ij}^{(k)}$  como 1 si existe un camino en el grafo  $G$  desde el vértice  $i$  al vértice  $j$  con todos los vértices intermedios en el conjunto  $f1; 2; \dots; k-1g$  y 0 en caso contrario. Construimos clausura transitiva  $\langle V, E' \rangle$  poniendo borde  $i, j \in E'$  si  $t_{ij}^{(n)} \neq 0$ . Una definición recursiva de  $t_{ij}^{(k)}$ , análoga a la recurrencia (25.5), es

$$t_{ij}^{(0)} = 1 \quad t_{ij}^{(k+1)} = \min(t_{ij}^{(k)}, t_{ij}^{(k)} \vee (t_{ij}^{(k)} \wedge t_{ij}^{(k)} \wedge t_{ij}^{(k)})) \quad (25.8)$$

Como en el algoritmo de Floyd-Warshall, calculamos las matrices  $T_{ij}^{(k)}$  en orden creciente de  $k$ .



	1000	1000	1000
T .0/ °	0111	0111	0111
	0110	0110	0111
	1011	1011	1011
	1000	1000	
T .3/ °	0111	1111	
	0111	1111	
	1111	1111	

Figura 25.5 Un gráfico dirigido y las matrices  $T_{.k/}$  calculadas por el algoritmo de cierre transitivo.

#### CIERRE-TRANSITIVO.G/

```

1 n D jG:Vj 2 sea
T .0/ .0/ Dt ser una nueva matriz nn
ij 3 para i D 1 a n 4
para j D 1 a n si i == jo .i;
5 j / 2 G:E .0/ D 1 t ij .0/ sino t
6 7 ij 8
k D 1 to n sea para D 0
9 D t ij .k/ ser una nueva matriz nn
10 para i D 1 a n
11 para j D 1 a n
12 tij.k/ .k1/.k1/ D t _ t ij ik t kj .k1/ ^
13 retorno T .n/

```

La figura 25.5 muestra las matrices  $T_{.k/}$  calculadas por el procedimiento de CIERRE TRANSITIVO en un gráfico de muestra. El procedimiento de CIERRE TRANSITIVO , como el algoritmo de Floyd-Warshall, se ejecuta en un tiempo de  $,n^3/$ . En algunas computadoras, sin embargo, las operaciones lógicas en valores de un solo bit se ejecutan más rápido que las operaciones aritméticas en palabras enteras de datos. Además, debido a que el algoritmo de cierre transitivo directo usa solo valores booleanos en lugar de valores enteros, su requisito de espacio es menor

que el algoritmo de Floyd-Warshall por un factor correspondiente al tamaño de una palabra de almacenamiento informático.

### Ejercicios

#### 25.2-1

Ejecute el algoritmo de Floyd-Warshall en el gráfico dirigido ponderado de la figura 25.2. Muestre la matriz  $D_k$  que resulta para cada iteración del ciclo externo.

#### 25.2-2

Muestre cómo calcular la clausura transitiva usando la técnica de la Sección 25.1.

#### 25.2-3

Modifique el procedimiento FLOYD-WARSHALL para calcular las matrices  $\dots.k$  de acuerdo con las ecuaciones (25.6) y (25.7). Demuestre rigurosamente que para todo  $i \in V$  el subgrafo predecesor  $G_i$  es un árbol de caminos más cortos con raíz  $i$ . (Sugerencia: para mostrar que  $D_k$  es acíclico, primero demuestre que  $G_i$  es acíclico con la definición de  $D_k$ . Luego, adapte la prueba del Lema 24.16.)

#### 25.2-4

Como aparece arriba, el algoritmo de Floyd-Warshall requiere un espacio  $n^3$ , ya que calculamos  $d_{ik}$  para  $i, j, k \in V$ ; norte. Demuestre que el siguiente procedimiento, que simplemente elimina todos los superíndices, es correcto y, por lo tanto, solo se requiere el espacio  $n^2$ .

```

FLOYD-WARSHALL0 .W / 1
n D W: filas 2 DDW
3 para k D 1
a n
4      para i D 1 a n
5      para j D 1 a n dij
6      D min .dij ; dik c dkj /
7 vuelta D

```

#### 25.2-5

Supongamos que modificamos la forma en que la ecuación (25.7) maneja la igualdad:

$$D_{ij}^{(k)} = \begin{cases} d_{ij} & \text{si } d_{ik} + d_{kj} < d_{ij} \\ d_{ik} + d_{kj} & \text{si } d_{ik} = d_{kj} \end{cases}$$

¿Es correcta esta definición alternativa de la matriz predecesora?

25.2-6

¿Cómo podemos usar la salida del algoritmo de Floyd-Warshall para detectar la presencia de un ciclo de peso negativo?

25.2-7

Otra forma de reconstruir las rutas más cortas en el algoritmo de Floyd-Warshall utiliza para  $i, j, k \in V$ ; intermedio  $k$  con el número más pequeño de pasos más corto de  $i$  a  $j$  en el que todos los vértices intermedios están en el conjunto  $\{1, 2, \dots, k\}$ . Proporcione una formulación recursiva para el procedimiento WARSHALL para calcular el procedimiento PARES-RUTA MÁS CORTA para tomar la matriz  $D^k_{ij}$ , modifique los valores de matriz  $D$  como entrada.

$D^k_{ij}$  FLOYD y reescriba PRINT-ALL

$D^{n-k}_{ij}$

¿En qué se parece la matriz  $D^k$  a la tabla s del problema de multiplicación de cadenas de matrices de la sección 15.2?

25.2-8

Proporcione un algoritmo O( $V^2E$ )-tiempo para calcular la clausura transitiva de un grafo dirigido  $G = (V, E)$ .

25.2-9

Suponga que podemos calcular la clausura transitiva de un grafo acíclico dirigido en  $O(V^2)$  tiempo, donde  $f$  es una función monótonamente creciente de  $|V|$  y  $|E|$ .

Muestre que el tiempo para calcular la clausura transitiva  $G = (V, E)$  de un grafo general dirigido  $G = (V, E)$  es entonces  $O(V^2 + |E|)$ .

### 25.3 Algoritmo de Johnson para grafos dispersos

El algoritmo de Johnson encuentra los caminos más cortos entre todos los pares en  $O(V^2 \lg VC VE)$  tiempo  $O(V^2)$ . Para gráficos dispersos, es asintóticamente más rápido que el cuadrado repetido de matrices o el algoritmo de Floyd-Warshall. El algoritmo devuelve una matriz de pesos de ruta más corta para todos los pares de vértices o informa que el gráfico de entrada contiene un ciclo de peso negativo. El algoritmo de Johnson utiliza como subrutinas tanto el algoritmo de Dijkstra como el algoritmo de Bellman-Ford, que se describen en el capítulo 24.

El algoritmo de Johnson utiliza la técnica de reponderación, que funciona de la siguiente manera. Si todos los pesos de las aristas  $w$  en un gráfico  $G = (V, E)$  son no negativos, podemos encontrar caminos más cortos entre todos los pares de vértices ejecutando el algoritmo de Dijkstra una vez desde cada vértice; con la cola de prioridad mínima del montón de Fibonacci, el tiempo de ejecución de este es  $O(V^2 \lg VC VE)$ . Si el algoritmo de todos los pares es  $O(V^2)$  tiene bordes de peso negativo pero no ciclos de peso negativo, simplemente calculamos un nuevo conjunto de pesos de borde no negativos

que nos permite utilizar el mismo método. El nuevo conjunto de pesos de borde  $w_y$  debe satisfacer dos propiedades importantes:

1. Para todos los pares de vértices  $u; 2$  voltios un camino  $p$  es el camino más corto desde  $u$  hasta usar la función de ponderación  $w$  si y solo si  $p$  es también el camino más corto desde  $u$  hasta usar la función de peso  $w_y$ .
2. Para todos los bordes  $.u; /$ , el nuevo peso  $w_u;$  / es no negativo.

Como veremos en un momento, podemos preprocesar  $G$  para determinar la nueva función de peso  $w_y$  en  $O.VE/$  tiempo.

Preservar los caminos más cortos al volver a ponderar

El siguiente lema muestra cuán fácilmente podemos volver a ponderar los bordes para satisfacer la primera propiedad anterior. Usamos  $w$  para denotar los pesos de la ruta más corta derivados de la función de ponderación  $w_y$  y  $w$  para denotar los pesos de la ruta más corta derivados de la función de ponderación  $w_y$ .

**Lema 25.1 (La reponderación no cambia los caminos más cortos)**

Dado un grafo dirigido y ponderado  $GD .V; E/$  con función de peso  $w$   $WE ! R$ , deja  $h$   $WV! R$  sea cualquier función que mapee vértices a números reales. Para cada arista  $.u; / 2 E,$

define  $w_u;$  /  $D w_u;$  /  $C h_u/ h./ : Sea p$  (25.9)

$D h_0; 1;:::k$  cualquier camino de vértice a vértice  $k$ . Entonces  $p$  es el camino más corto desde con función  $w$  si y sólo si es el camino más corto a  $k D$  si  $w_p/$  Además,  $G$  tiene un ciclo de peso negativo si y solo si  $w$  si y solo  $w$  si y solo si  $G$  tiene un ciclo de peso negativo usando la función de peso  $w_y$ .

Prueba Comenzamos mostrando

que  $w_p/ D w_p/ C h_0/ h_k/ :$  (25.10)

Tenemos

$$\begin{array}{c}
 & k \\
 & \downarrow \\
 wp/ & DX w. i1; i/ \\
 & \quad iD1 \\
 & \quad k \\
 & \quad \downarrow \\
 & DX .w.i1; i/ C h.i1/ hola// \\
 & \quad iD1 \\
 & \quad k \\
 & \quad \downarrow \\
 & DX w.i1; i/ C h.0/ h_k/ \\
 & \quad iD1
 \end{array}$$

(porque la suma se eleva)

$D wp/ C h.0/ h_k/ :$

Por lo tanto, cualquier camino desde  $p$  tiene  $w_p / D_{p,p} / C_{h,0} / h_k /$ . Sea  $a_k$  porque  $h_{0,0}$  y  $h_k /$  dependen del camino, si un camino desde  $k$  es más corto que otro usando la función  $w$ , entonces también es más corto usando  $w_y$ . Así,  $D_{y,0} / k / w_y / D_{1,0} / k /$  si y solo si  $w_y /$  mostramos que  $G$  tiene un ciclo de peso . Finalmente,

negativo usando la función de peso  $w$  si y solo si  $G$  tiene un ciclo de peso negativo usando la función de peso  $w_y$ . Considere cualquier ciclo  $c$   $D_{h,0} / 1 / \dots / k /$ , donde  $k$ . Por la ecuación (25.10),

$$W_c / D_{wc} / C_{h,0} / h_k / D_{wc} / ;$$

y por lo tanto  $c$  tiene peso negativo usando  $w$  si y solo si tiene peso negativo usando  $w_y$ .

■

### Producir pesos no negativos mediante la reponderación

Nuestro siguiente objetivo es asegurarnos de que se cumple la segunda propiedad: queremos  $w_{u,v} /$  ser no negativo para todas las aristas  $u,v \in E$ . Dado un grafo dirigido ponderado  $G = (V, E)$  con función de peso  $w : V \times V \rightarrow \mathbb{R}$ , hacemos un nuevo grafo  $G_0 = (V_0, E_0)$ , donde  $V_0 = \{s\} \cup V \cup \{t\}$  para algún nuevo vértice  $s$  y  $t$  y  $E_0 = \{(s, u) | u \in V\} \cup \{(v, t) | v \in V\}$ .

Extendemos la función de ponderación  $w$  de modo que  $w_{s,u} = 0$  para todos los  $u \in V$ . Tenga en cuenta que debido a que  $s$  no tiene bordes que entren en él, ninguna ruta más corta en  $G_0$  que no sea la fuente  $s$  contiene  $s$ . Además,  $G_0$  no tiene ciclos de peso negativo si y solo si  $G$  no tiene ciclos de peso negativo. La figura 25.6(a) muestra la gráfica  $G_0$  correspondiente a la gráfica  $G$  de la figura 25.1.

Ahora suponga que  $G$  y  $G_0$  no tienen ciclos de peso negativo. Definamos. Por la  $h_i / D_{i,s} = 0$  para todos los  $i \in V$  tenemos desigualdad triangular (Lema 24.10),  $h_u / C_{wu} = 0$  para todos los bordes  $u \in E$ . Así, si definimos los nuevos pesos  $w_y$  reponeránlos de acuerdo con la ecuación (25.9), tenemos  $D_{wu} = h_u / 0$ , y hemos  $w_y = h_u / C_{wu}$ . La satisfecho la segunda propiedad.

Figura 25.6(b) muestra el gráfico  $G_0$  de la Figura 25.6(a) con bordes reponderados.

### Cálculo de las rutas más cortas de todos los pares

El algoritmo de Johnson para calcular los caminos más cortos de todos los pares utiliza el algoritmo de Bellman-Ford (sección 24.1) y el algoritmo de Dijkstra (sección 24.3) como subrutinas. Asume implícitamente que los bordes se almacenan en listas de adyacencia. El algoritmo devuelve la matriz usual  $D = (D_{i,j})$ , donde  $D_{i,j} = \min_{k \in V} (D_{i,k} + D_{k,j})$ , o informa que el gráfico de entrada contiene un ciclo de peso negativo. Como es típico para un algoritmo de caminos más cortos de todos los pares, asumimos que los vértices están numerados de 1 a  $|V|$ .

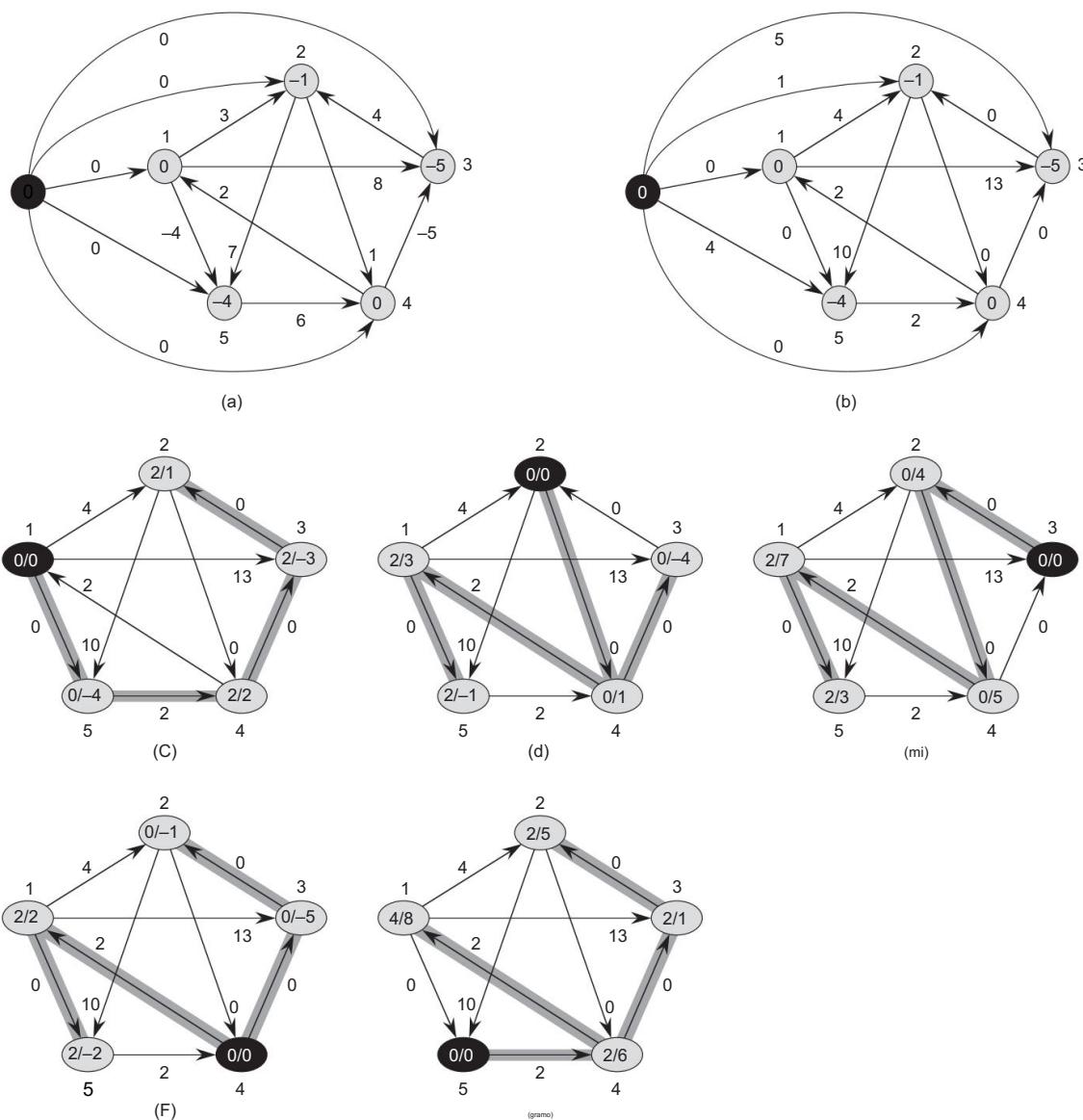


Figura 25.6 El algoritmo de caminos más cortos de todos los pares de Johnson se ejecuta en el gráfico de la figura 25.1. Los números de vértice aparecen fuera de los vértices. (a) El gráfico  $G_0$  con la función de peso original  $w$ . El nuevo vértice  $s$  es negro. Dentro de cada vértice está  $h./D\ i.s.; /$ . (b) Despues de volver a ponderar cada borde  $.u;$  / con función de peso  $wu;$  /  $D\ wu;$  /  $C\ hu/\ h./$ . (c)-(g) El resultado de ejecutar el algoritmo de Dijkstra en cada vértice de  $G$  usando la función de ponderación  $wy$ . En cada parte, el vértice de origen  $u$  es negro y los bordes sombreados están en el árbol de caminos más cortos calculado por el algoritmo. Dentro de cada vértice están los valores  $i.u; / y i.u; /$ , separados por una barra. El valor del  $D\ i.u; /$  es igual a  $C\ h./ hu/\ i. / tu;$

JOHNSON.G; w/ 1

```
calcule G0 G0 :E , donde G0 :V DG:V [ fsg,
DG:E [ fs; / W2G :Vg, y ws; / D 0 para todo 2
G:V 2 si BELLMAN-FORD.G0 ;w;
s/ == FALSO 3 imprime "el gráfico de entrada
contiene un ciclo de peso negativo" 4 más para cada vértice 2 G0 :V 5
```

```
fije h./ al valor de i.s; / calculada por
el algoritmo Bellman-Ford para cada arista .u; / 2
6      G0 :E wu; / D wu; / C hu/ h./ sea
7          DD .du / una nueva matriz nn para cada
8      vértice u 2 G:V
9
10     ejecutar DIJKSTRA.G; w; u/ para calcular yi.u; / para todos los 2G:V
11     para cada vértice 2 G:V
12         du D i.u; / C h./ hu/ volver D
13
```

Este código simplemente realiza las acciones que especificamos anteriormente. La línea 1 produce G0 . La línea 2 ejecuta el algoritmo de Bellman-Ford en G0 con la función de peso w y el vértice fuente s. Si G0 y, por lo tanto, G, contiene un ciclo de peso negativo, la línea 3 informa el problema. Las líneas 4 a 12 asumen que G0 no contiene ciclos de peso negativo. Las líneas 4 y 5 ajustan h./ al peso del camino más corto i.s; / calculado por el algoritmo Bellman-Ford. Las líneas 6 y 7 calculan ritmo para todos 2 V ° los nuevos pesos wy. Para cada par de ver, el ciclo for de las líneas 9–12 te hace gracia; 2 voltios , calcula el peso del camino más corto i.u; / llamando al algoritmo de Dijkstra una vez desde cada vértice en V . La línea 12 almacena en la entrada de la matriz du el peso correcto del camino más corto i.u; /, calculado mediante la ecuación (25.10). Finalmente, la línea 13 devuelve la matriz D completa. La figura 25.6 muestra la ejecución del algoritmo de Johnson.

Si implementamos la cola de prioridad mínima en el algoritmo de Dijkstra por un tiempo de montón, el algoritmo de Johnson se ejecuta en  $^2$  Fibonacci lg V CVE/. El min binario más simple OV. La implementación del montón produce un tiempo de ejecución de O.VE lg V/, que sigue siendo asintóticamente más rápido que el algoritmo de Floyd-Warshall si el gráfico es disperso.

## Ejercicios

### 25.3-1

Utilice el algoritmo de Johnson para encontrar los caminos más cortos entre todos los pares de vértices en la gráfica de la figura 25.2. Muestre los valores de h y wy calculados por el algoritmo.

## 25.3-2

¿ Cuál es el propósito de sumar el nuevo vértice s a V , dando V<sup>o</sup>?

## 25.3-3

Supongamos que  $w_{u,v} = 0$  para todos los bordes  $u, v \in E$ . ¿Cuál es la relación entre las funciones de peso  $w$  y  $w'$ ?

## 25.3-4

El profesor Greenstreet afirma que existe una forma más sencilla de volver a ponderar los bordes que el método utilizado en el algoritmo de Johnson. Dejando  $w'_{u,v} = \min\{w_{u,v}, w_{v,u}\}$ , simplemente defina  $w'_{u,v} = 0$  para todos los bordes  $u, v \in E$ . ¿Qué tiene de malo el método de reponderación del profesor?

## 25.3-5

Suponga que ejecutamos el algoritmo de Johnson en un gráfico dirigido  $G$  con una función de peso  $w$ . Muestre que si  $G$  contiene un ciclo de peso 0  $c$ , entonces  $w'_{u,v} = 0$  para cada arista  $u, v \in C$ .

## 25.3-6

El profesor Michener afirma que no hay necesidad de crear un nuevo vértice fuente en la línea 1 de JOHNSON. Afirma que, en cambio, podemos usar  $G_0$  DG y seamos cualquier vértice. Dé un ejemplo de un gráfico  $G$  dirigido y ponderado para el cual la incorporación de la idea del profesor en JOHNSON provoque respuestas incorrectas. Luego demuestre que si  $G$  está fuertemente conectado (cada vértice es accesible desde cualquier otro vértice), los resultados devueltos por JOHNSON con la modificación del profesor son correctos.

## Problemas

## 25-1 Cierre transitivo de un grafo dinámico Supongamos

que deseamos mantener el cierre transitivo de un grafo dirigido  $G$ . A medida que insertamos aristas en  $E$ . Es decir, después de que se haya insertado cada arista, queremos actualizar el cierre transitivo de las aristas insertadas hasta el momento. Suponga que el grafo  $G$  no tiene aristas inicialmente y que representamos el cierre transitivo como una matriz booleana.

a. Mostrar cómo actualizar la clausura transitiva  $G$  de un grafo  $G$  en el momento en que se agrega un nuevo borde a  $G$ .

b. Dé un ejemplo de un gráfico  $G$  y una arista  $e$  tal que se requiera  $G$  en el tiempo para actualizar el cierre transitivo después de la inserción de  $e$  en  $G$ , sin importar qué algoritmo se use.

C. Describa un algoritmo eficiente para actualizar el cierre transitivo a medida que se insertan bordes en el gráfico. Para cualquier secuencia de  $n$  inserciones, su algoritmo debe ejecutarse en el tiempo total  $P_n \leq D \cdot O(V^3)$ , donde  $D$  es el tiempo para actualizar el cierre transitivo al insertar el  $i$ -ésimo borde. Demuestre que su algoritmo alcanza este límite de tiempo.

#### 25-2 Caminos más cortos en grafos -densos

Un grafo  $G = (V, E)$  es -denso si  $|E| \geq C|V|^2$  para alguna constante en el rango  $0 < C \leq 1$ . Usando montones mínimos  $d$ -arios (vea el problema 6-2) en algoritmos de caminos más cortos en grafos -densos, podemos coincidir con los tiempos de ejecución de los algoritmos basados en el montón de Fibonacci sin utilizar una estructura de datos tan complicada.

- ¿Cuáles son los tiempos de ejecución asintóticos para INSERT, EXTRACT-MIN y DECREASE-KEY, en función de  $d$  y el número  $n$  de elementos en un  $d$ -ary min-heap? ¿Cuáles son estos tiempos de ejecución si elegimos  $d = \sqrt{n}$  para alguna constante  $0 < C \leq 1$ ? Compare estos tiempos de ejecución con los costos amortizados de estas operaciones para un montón de Fibonacci.
- Muestre cómo calcular las rutas más cortas desde una sola fuente en un gráfico dirigido -denso  $G = (V, E)$  sin bordes de peso negativo en el tiempo  $O(|E|)$ . (Sugerencia: elige  $d$  como una función de  $|V|$ ).
- Muestre cómo resolver el problema de los caminos más cortos de todos los pares en un grafo dirigido denso  $G = (V, E)$  sin flancos de peso negativo en tiempo  $O(|V|^2)$ .
- Muestre cómo resolver el problema de los caminos más cortos de todos los pares en  $O(|V|^2)$  tiempo en un gráfico dirigido -denso  $G = (V, E)$  que puede tener bordes de peso negativo pero no tiene ciclos de peso negativo.

#### Notas del capítulo

Lawler [224] tiene una buena discusión sobre el problema de los caminos más cortos de todos los pares, aunque no analiza soluciones para grafos dispersos. Atribuye el algoritmo de multiplicación de matrices al folclor. El algoritmo de Floyd-Warshall se debe a Floyd [105], quien se basó en un teorema de Warshall [349] que describe cómo calcular el cierre transitivo de matrices booleanas. El algoritmo de Johnson está tomado de [192].

Varios investigadores han proporcionado algoritmos mejorados para calcular los caminos más cortos a través de la multiplicación de matrices. Fredman [111] muestra cómo resolver el problema de los caminos más cortos de todos los pares usando  $O(|V|^2 \log |V|)$  comparaciones entre sumas de aristas

pondera y obtiene un algoritmo que se ejecuta en  $O(V^3 \lg \lg V = \lg V / 1=3)$  tiempo, que es ligeramente mejor que el tiempo de ejecución del algoritmo de Floyd-Warshall. Han [159] redujo el tiempo de ejecución a  $O(V^3 \lg \lg V = \lg V / 5=4)$ . Otra línea de investigación demuestra que podemos aplicar algoritmos para la multiplicación rápida de matrices (véanse las notas del capítulo 4) al problema de los caminos más cortos de todos los pares. Sea  $O(n!)$  el tiempo de ejecución del algoritmo más rápido para multiplicar  $n$  matrices; actualmente  $! < 2:376$  [78]. Galil y Margalit [123, 124] y Seidel [308] diseñaron algoritmos que resuelven el problema de los caminos más cortos de todos los pares en gráficos no dirigidos y no ponderados en  $O(V! pV)$  tiempo, donde  $p_n$  denota una función particular que es polilogarítmicamente acotado en  $n$ . En gráficos densos, estos algoritmos son más rápidos que el tiempo  $O(VE)$  necesario para realizar búsquedas  $jV$  en anchura. Varios investigadores han ampliado estos resultados para dar algoritmos para resolver el problema de los caminos más cortos de todos los pares en gráficos no dirigidos en los que los pesos de los bordes son números enteros en el rango  $f_1, f_2, \dots, f_W$ . El algoritmo asintóticamente más rápido, de Shoshan y Zwick [316], se ejecuta en el tiempo  $O(WV \lg V W)$ .

Karger, Koller y Phillips [196] e, independientemente, McGeoch [247] han dado un límite de tiempo que depende de  $E$ , el conjunto de aristas en  $E$  que participan en algún camino más corto. Dado un gráfico con pesos de borde no negativos, sus algoritmos se ejecutan en  $O(VE CV^2 \lg V)$  tiempo y mejora al ejecutar el algoritmo de Dijkstra  $jV$  veces cuando  $jEj D oE$ .

Baswana, Hariharan y Sen [33] examinaron algoritmos decrementales para mantener los caminos más cortos de todos los pares y la información de cierre transitivo. Los algoritmos decrementales permiten una secuencia de supresiones de bordes y consultas entremezcladas; en comparación, el problema 25-1, en el que se insertan aristas, pide un algoritmo incremental. Los algoritmos de Baswana, Hariharan y Sen son aleatorios y, cuando existe un camino, su algoritmo de cierre transitivo puede fallar al informarlo con probabilidad  $1=nc$  para un  $c>0$  arbitrario. Los tiempos de consulta son  $O(1/c)$  con alta probabilidad. Para cierre transitivo, el tiempo amortizado para cada actualización es  $O(V^4=3 \lg 1=3 V)$ . Para las rutas más cortas de todos los pares, los tiempos de actualización dependen de las consultas. Para las consultas que solo dan los pesos de la ruta más corta, el tiempo amortizado por actualización es  $O(V^3 = E \lg 2 V)$ . Para informar la ruta más corta real, el tiempo de actualización amortizado es  $\min(O(V^3 = 2plg V), O(V^3 = E \lg 2 V))$ . Demetrescu e Italiano [84] mostraron cómo manejar las operaciones de actualización y consulta cuando los bordes se insertan y eliminan, siempre que cada borde dado tenga un rango limitado de valores posibles extraídos de los números reales.

Aho, Hopcroft y Ullman [5] definieron una estructura algebraica conocida como “semiring cerrado”, que sirve como marco general para resolver problemas de trayectoria en grafos dirigidos. Tanto el algoritmo de Floyd-Warshall como el algoritmo de cierre transitivo de la Sección 25.2 son instancias de un algoritmo de todos los pares basado en semianillos cerrados. Maggs y Plotkin [240] mostraron cómo encontrar árboles de expansión mínimos utilizando un semianillo cerrado.

Así como podemos modelar un mapa de carreteras como un gráfico dirigido para encontrar el camino más corto de un punto a otro, también podemos interpretar un gráfico dirigido como una "red de flujo" y usarlo para responder preguntas sobre flujos de materiales. Imagine un material que recorre un sistema desde una fuente, donde se produce el material, hasta un sumidero, donde se consume. La fuente produce el material a una tasa constante y el sumidero consume el material a la misma tasa. El "flujo" del material en cualquier punto del sistema es intuitivamente la velocidad a la que se mueve el material.

Las redes de flujo pueden modelar muchos problemas, incluidos líquidos que fluyen a través de tuberías, piezas a través de líneas de ensamblaje, corriente a través de redes eléctricas e información a través de redes de comunicación.

Podemos pensar en cada borde dirigido en una red de flujo como un conducto para el material. Cada conducto tiene una capacidad establecida, dada como la tasa máxima a la que el material puede fluir a través del conducto, como 200 galones de líquido por hora a través de una tubería o 20 amperios de corriente eléctrica a través de un cable. Los vértices son uniones de conductos y, aparte de la fuente y el sumidero, el material fluye a través de los vértices sin acumularse en ellos. En otras palabras, la velocidad a la que el material entra en un vértice debe ser igual a la velocidad a la que sale del vértice. Llamamos a esta propiedad "conservación del flujo" y es equivalente a la ley de corriente de Kirchhoff cuando el material es corriente eléctrica.

En el problema de flujo máximo, deseamos calcular la mayor velocidad a la que podemos enviar material desde la fuente hasta el sumidero sin violar ninguna restricción de capacidad. Es uno de los problemas más simples relacionados con las redes de flujo y, como veremos en este capítulo, este problema se puede resolver mediante algoritmos eficientes.

Además, podemos adaptar las técnicas básicas utilizadas en los algoritmos de flujo máximo para resolver otros problemas de flujo de red.

Este capítulo presenta dos métodos generales para resolver el problema de flujo máximo. La Sección 26.1 formaliza las nociones de redes de flujo y flujos, definiendo formalmente el problema de flujo máximo. La sección 26.2 describe el método clásico de Ford y Fulkerson para encontrar flujos máximos. Una aplicación de este método,

encontrar una coincidencia máxima en un gráfico bipartito no dirigido aparece en la Sección 26.3. La sección 26.4 presenta el método push-relabel, que subyace a muchos de los algoritmos más rápidos para problemas de flujo de red. La Sección 26.5 cubre el algoritmo de "reetiquetado al frente", una implementación particular del método de reetiquetado por inserción que se ejecuta en el tiempo  $O(V^3)$ . Aunque este algoritmo no es el algoritmo más rápido conocido, ilustra algunas de las técnicas utilizadas en los algoritmos asintóticamente más rápidos y es razonablemente eficiente en la práctica.

## 26.1 Redes de flujo

En esta sección, damos una definición teórica de grafos de redes de flujo, analizamos sus propiedades y definimos con precisión el problema de flujo máximo. También presentamos algunas notaciones útiles.

### Redes de flujo y flujos

Una red de flujo  $G = (V, E)$  es un grafo dirigido en el que cada arista  $u \in E$  tiene una capacidad no negativa  $c(u) \geq 0$ . Requerimos además que si  $E$  contiene una arista  $u \in E$ , entonces no hay borde  $v \in u$  en sentido inverso. (Veremos en breve cómo solucionar esta restricción). Si  $u \in E$ , entonces por conveniencia definimos  $c(u) = \infty$ , y no permitimos los bucles automáticos. Distinguimos dos vértices en una red de flujo: una fuente  $s$  y un sumidero  $t$ . Por conveniencia, asumimos que cada vértice se encuentra en algún camino desde la fuente hasta el sumidero. Es decir, para todo  $v \in V$ , cada vértice  $v$  la red de flujo contiene un camino  $s \rightarrow v \rightarrow t$ . Por tanto, el grafo es conexo y, dado que cada vértice distinto de  $s$  tiene al menos una arista de entrada,  $\sum_{u \in E} c(u) \geq 1$ . La figura 26.1 muestra un ejemplo de una red de flujo.

Ahora estamos listos para definir los flujos de manera más formal. Sea  $G = (V, E)$  ser una red de flujo con una función de capacidad  $c$ . Sea  $s$  la fuente de la red y  $t$  el sumidero. Un flujo en  $G$  es una función de valor real  $f: V \rightarrow \mathbb{R}$  que satisface las siguientes dos propiedades:

Restricción de capacidad: Para todo  $u \in E$ , queremos  $0 \leq f(u) \leq c(u)$ .

Conservación del flujo: para todos los  $u \in E$ , queremos

$$\sum_{v \in N^+(u)} f(v) = \sum_{v \in N^-(u)} f(v)$$

Cuando  $u \in E$ , no puede haber flujo de  $u$  a  $t$ , y  $f(u) = 0$ .

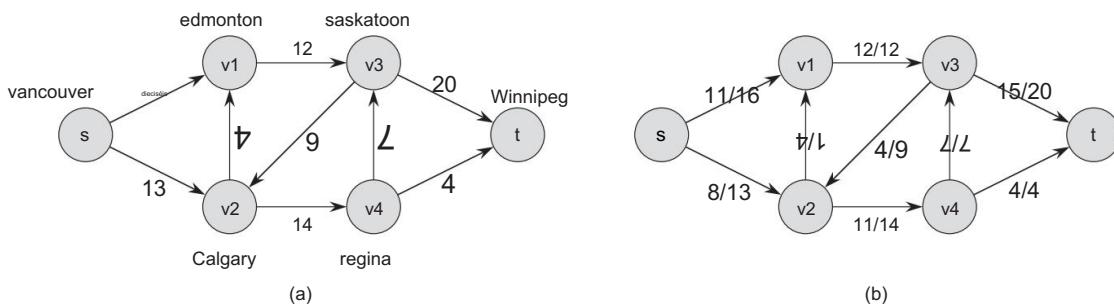


Figura 26.1 (a) Una red de flujo GD .V; E/ por el problema de camiones de Lucky Puck Company. La fábrica de Vancouver es la fuente y el almacén de Winnipeg es el sumidero. La empresa envía discos a través de ciudades intermedias, pero solo cu; / cajas por día pueden ir de ciudad a ciudad. Cada borde está etiquetado con su capacidad. (b) Un flujo f en G con valor  $\sum_j f(j)$ . Cada borde  $(u, v)$  está etiquetado por  $f(u, v) = c_{uv}$ . La notación de barra simplemente separa el flujo y la capacidad; no indica división.

Llamamos a la cantidad no negativa  $f(u)$  el flujo del vértice  $u$  al vértice. El valor  $f(j)$  de un flujo  $f$  se define como

if j DX f .s; / Xf.; s/ ;  
 2V                    2V

es decir, el flujo total que sale de la fuente menos el flujo que entra en la fuente. (Aquí, la notación  $\sum_j$  denota valor de flujo, no valor absoluto o cardinalidad). Por lo general, una red de flujo no tendrá ningún borde hacia la fuente, y el flujo hacia la fuente, dado por la suma  $\sum_{f_j(s)}$ , será 0. Sin embargo, lo incluimos porque cuando introduzcamos las redes residuales más adelante en este capítulo, el flujo hacia la fuente será significativo. En el problema de flujo máximo, tenemos una red de flujo  $G$  con fuente  $s$  y sumidero  $t$ , y deseamos encontrar un flujo de valor máximo.

Antes de ver un ejemplo de un problema de flujo de red, exploremos brevemente la definición de flujo y las dos propiedades de flujo. La restricción de capacidad simplemente dice que el flujo de un vértice a otro debe ser no negativo y no debe exceder la capacidad dada. La propiedad de conservación del flujo dice que el flujo total hacia un vértice que no sea la fuente o el sumidero debe ser igual al flujo total que sale de ese vértice; de manera informal, "el flujo que entra es igual al flujo que sale".

## Un ejemplo de flujo

Una red de flujo puede modelar el problema de camiones que se muestra en la figura 26.1(a). Lucky Puck Company tiene una fábrica (fuente s) en Vancouver que fabrica discos de hockey, y tiene un depósito (sink t) en Winnipeg que los almacena. Afortunado

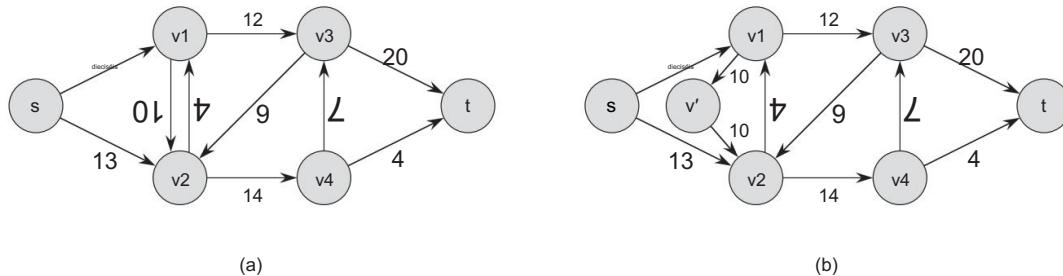


Figura 26.2 Conversión de una red con aristas antiparalelas a una equivalente sin aristas antiparalelas. (a) Una red de flujo que contiene ambos bordes .1; 2/ y .2; 1/. (b) Una red equivalente sin bordes antiparalelos. Añadimos las nuevas  $\overrightarrow{e_3}$  y  $\overleftarrow{e_3}$ , y reemplazamos edge .1; 2/ por el par de / aristas de los bordes  $e_1$  y  $e_2$  con la misma capacidad que .1; 2/.

Puck alquila espacio en camiones de otra empresa para enviar los discos de la fábrica al almacén. Debido a que los camiones recorren rutas específicas (bordes) entre ciudades (vértices) y tienen una capacidad limitada, Lucky Puck puede realizar envíos como máximo; / cajas por día entre cada par de ciudades u y en la Figura 26.1(a). Lucky Puck no tiene control sobre estas rutas y capacidades, por lo que la empresa no puede alterar la red de flujo que se muestra en la figura 26.1(a). Necesitan determinar la mayor cantidad  $p$  de cajas por día que pueden enviar y luego producir esta cantidad, ya que no tiene sentido producir más discos de los que pueden enviar a su almacén.

Lucky Puck no se preocupa por el tiempo que tarda un disco determinado en llegar desde la fábrica hasta el almacén; sólo les importa que  $p$  cajas por día salgan de la fábrica y que  $p$  cajas por día lleguen al almacén.

Podemos modelar el "flujo" de envíos con un flujo en esta red porque la cantidad de cajas enviadas por día de una ciudad a otra está sujeta a una restricción de capacidad. Además, el modelo debe obedecer a la conservación del flujo, ya que en un estado estacionario, la velocidad a la que entran los discos en una ciudad intermedia debe ser igual a la velocidad a la que salen. De lo contrario, las cajas se acumularían en las ciudades intermedias.

## Problemas de modelado con aristas antiparalelas

Suponga que la empresa de camiones le ofrece a Lucky Puck la oportunidad de arrendar espacio para 10 cajas en camiones que van de Edmonton a Calgary. Parecería natural agregar esta oportunidad a nuestro ejemplo y formar la red que se muestra en la figura 26.2(a). Sin embargo, esta red tiene un problema: viola nuestra suposición original de que si un borde .1; 2/ 2 E, luego .2; 1/ 62 E. A las dos aristas las llamamos .1; 2/ y .2; 1/ antiparalelo. Por tanto, si deseamos modelar un problema de flujo con aristas antiparalelas, debemos transformar la red en una red equivalente que no contenga

aristas antiparalelas. La Figura 26.2(b) muestra esta red equivalente. Elegimos una de las dos aristas antiparalelas, en este caso  $.1; 2/$ , y divídalo agregando un nuevo vértice  $^0$  y  $.0 ; 2/$ . reemplazando el borde  $.1; 2/$  con el par de aristas  $.1;$   $.0$ . También establecemos la capacidad de ambos bordes nuevos a la capacidad del borde original. La red resultante satisface la propiedad de que si un borde está en la red, el borde inverso no lo está. El ejercicio 26.1-1 le pide que demuestre que la red resultante es equivalente a la original.

Por lo tanto, vemos que un problema de flujo del mundo real podría modelarse de manera más natural mediante una red con bordes antiparalelos. Sin embargo, será conveniente deshabilitar los bordes antiparalelos, por lo que tenemos una forma sencilla de convertir una red que contiene bordes antiparalelos en una red equivalente sin bordes antiparalelos.

#### Redes con múltiples fuentes y sumideros

Un problema de flujo máximo puede tener varias fuentes y sumideros, en lugar de solo uno de cada uno. The Lucky Puck Company, por ejemplo, en realidad podría tener un conjunto de  $m$  fábricas  $s_1; s_2; \dots; s_m$  y un conjunto de  $n$  almacenes  $t_1; t_2; \dots; t_n$ , como se muestra en la figura 26.3(a). Afortunadamente, este problema no es más difícil que el flujo máximo ordinario.

Podemos reducir el problema de determinar un flujo máximo en una red con múltiples fuentes y múltiples sumideros a un problema ordinario de flujo máximo. La figura 26.3(b) muestra cómo convertir la red (a) en una red de flujo ordinario con una sola fuente y un solo sumidero. Agregamos una superfuente  $s$  y agregamos un borde dirigido  $.s; s_i/$  con capacidad  $c_{s_i}$ ;  $s_i/ D_i$  para cada  $i$ ;  $D_1; D_2; \dots; D_m$  metro. También creamos un nuevo supersink  $t$  y agregamos un borde dirigido  $.t_i; t/$  con capacidad  $c_{t_i}$ ;  $t/ D_i$  para cada  $i$ ;  $D_1; D_2; \dots; D_m$  norte. Intuitivamente, cualquier flujo en la red de (a) corresponde a un flujo en la red de (b), y viceversa. La fuente única  $s$  simplemente proporciona tanto flujo como se deseé para las fuentes múltiples  $s_i$ , y el sumidero único  $t$  también consume tanto flujo como se deseé para los sumideros múltiples  $t_i$ . El ejercicio 26.1-2 le pide que demuestre formalmente que los dos problemas son equivalentes.

#### Ejercicios

##### 26.1-1

Muestre que dividir un borde en una red de flujo produce una red equivalente. Más formalmente, suponga que la red de flujo  $G$  contiene el borde  $.u; /$ , y creamos una nueva red de flujo  $G_0$  creando un nuevo vértice  $x$  y reemplazando  $.u; /$  por nuevas aristas  $.u; x/$  y  $.x; /$  con  $c_u$ ;  $x/D_c x; / D_{cu}; /$ .

Muestre que un caudal máximo en  $G_0$  tiene el mismo valor que un caudal máximo en  $G$ .

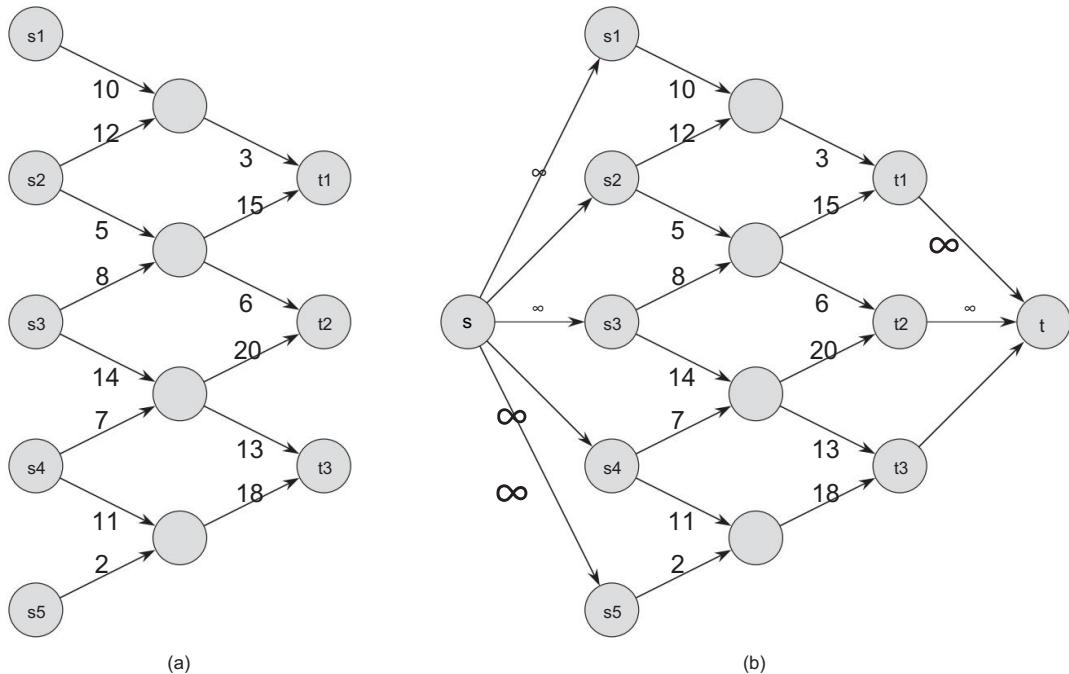


Figura 26.3 Conversión de un problema de flujo máximo de múltiples fuentes y múltiples sumideros en un problema con una sola fuente y un solo sumidero. (a) Una red de flujo con cinco fuentes SD  $s_1; s_2; s_3; s_4; s_5$  y tres lavabos TD  $t_1; t_2; t_3$ . (b) Una red de flujo equivalente de fuente única y sumidero único. Agregamos una superfuente  $s$  y un borde con capacidad infinita de  $s$  a cada una de las múltiples fuentes. También agregamos un supersink  $t$  y un borde con capacidad infinita desde cada uno de los múltiples sumideros hasta  $t$ .

### 26.1-2

Extienda las propiedades y definiciones de flujo al problema de múltiples fuentes y múltiples sumideros. Muestre que cualquier flujo en una red de flujo de múltiples fuentes y múltiples sumideros corresponde a un flujo de valor idéntico en la red de una sola fuente y un solo sumidero obtenida al agregar una superfuente y un supersumidero, y viceversa.

### 26.1-3

Suponga que una red de flujo GD .V; E/ viola la suposición de que la red contiene un camino  $s \rightarrow t$  para todos los vértices  $2 V$ . Sea  $u$  un vértice para el que no existe camino  $s \rightarrow u \rightarrow t$ . Muestre que debe existir un flujo máximo  $f$  en  $G$  tal que  $f(u) = 0$  para todos los vértices  $2 V$ .

## 26.1-4

Sea  $f$  un flujo en una red y sea  $\lambda$  un número real. El producto de flujo escalar, denotado  $\lambda f$ , es una función de  $VV$  a  $R$  definida por

$$\lambda f / u; / D \lambda f . u; / :$$

Demostrar que los flujos en una red forman un conjunto convexo. Es decir, demuestre que si  $f_1$  y  $f_2$  son flujos, entonces también lo es  $\lambda f_1 + (1 - \lambda) f_2$  para todo  $\lambda$  en el rango  $0 \leq \lambda \leq 1$ .

## 26.1-5

Expresese el problema de flujo máximo como un problema de programación lineal.

## 26.1-6

El profesor Adam tiene dos hijos que, lamentablemente, no se caen bien. El problema es tan grave que no sólo se niegan a caminar juntos a la escuela, sino que cada uno se niega a caminar sobre cualquier bloque que el otro niño haya pisado ese día.

Los niños no tienen problema en que sus caminos se crucen en una esquina. Afortunadamente tanto la casa del profesor como la escuela están en esquinas, pero más allá de eso no está seguro si será posible enviar a sus dos hijos a la misma escuela.

El profesor tiene un mapa de su ciudad. Muestre cómo formular el problema de determinar si sus dos hijos pueden ir a la misma escuela como un problema de flujo máximo.

## 26.1-7

Suponga que, además de las capacidades de borde, una red de flujo tiene capacidades de vértice. Es decir, cada vértice tiene un límite  $l_v$  sobre la cantidad de flujo que puede pasar. Mostrar cómo transformar una red de flujo  $G = (V, E)$  con capacidades de vértice en una red de flujo equivalente  $G' = (V', E')$  sin capacidades de vértice, tal que un flujo máximo en  $G'$  tiene el mismo valor que un flujo máximo en  $G$ . ¿Cuántos vértices y aristas tiene  $G'$ ?

## 26.2 El método Ford-Fulkerson

Esta sección presenta el método de Ford-Fulkerson para resolver el problema de flujo máximo. Lo llamamos un "método" en lugar de un "algoritmo" porque abarca varias implementaciones con diferentes tiempos de ejecución. El método Ford-Fulkerson depende de tres ideas importantes que trascienden el método y son relevantes para muchos algoritmos y problemas de flujo: redes residuales, caminos de aumento y cortes. Estas ideas son esenciales para el importante teorema de corte mínimo de flujo máximo (El teorema 26.6), que caracteriza el valor de un flujo máximo en términos de cortes de

la red de flujo. Terminamos esta sección presentando una implementación específica del método Ford-Fulkerson y analizando su tiempo de ejecución.

El método de Ford-Fulkerson incrementa iterativamente el valor del flujo. Empezamos con  $f(u) = 0$  para todo  $u \in V$ , dando un flujo inicial de valor 0. En cada iteración, aumentamos el valor de flujo en  $G$  encontrando una "ruta de aumento" en una "red residual" asociada  $G_f$ . Una vez que conocemos los bordes de un camino de aumento en  $G_f$ , podemos identificar fácilmente los bordes específicos en  $G$  para los que podemos cambiar el flujo para aumentar el valor del flujo. Aunque cada iteración del método de Ford-Fulkerson aumenta el valor del flujo, veremos que el flujo en cualquier borde particular de  $G$  puede aumentar o disminuir; Puede ser necesario disminuir el flujo en algunos bordes para permitir que un algoritmo envíe más flujo desde la fuente al sumidero. Aumentamos repetidamente el flujo hasta que la red residual no tenga más rutas de aumento. El teorema de corte mínimo de flujo máximo mostrará que al terminar, este proceso produce un flujo máximo.

```
FORD-FULKERSON-METHOD(G; s; t) / 1
```

```
    inicializa el flujo f a 0 / 2
```

```
    mientras exista un camino de aumento p en la red residual Gf / 3
```

```
        aumentar flujo f a lo largo
```

```
        de p / 4 retorno f
```

Para implementar y analizar el método Ford-Fulkerson, necesitamos introducir varios conceptos adicionales.

### Redes residuales

Intuitivamente, dada una red de flujo  $G$  y un flujo  $f$ , la red residual  $G_f$  consta de aristas con capacidades que representan cómo podemos cambiar el flujo en las aristas de  $G$ .

Un borde de la red de flujo puede admitir una cantidad de flujo adicional igual a la capacidad del borde menos el flujo en ese borde. Si ese valor es positivo, colocamos ese borde en  $G_f$  con una "capacidad residual" de  $c_f(u) = D_{uv} - f(u)$ .

Las únicas aristas de  $G$  que están en  $G_f$  son las que pueden admitir más flujo; esos bordes  $u$  tienen una capacidad residual  $c_f(u) > 0$ .

Sin embargo, la red residual  $G_f$  también puede contener aristas que no están en  $G$ . Como un algoritmo manipula el flujo, con el objetivo de aumentar el flujo total, es posible que deba disminuir el flujo en un borde particular. Para representar una posible disminución de un flujo positivo  $f(u)$  sobre una arista en  $G$ , colocamos una arista  $u$  en  $G_f$  con capacidad residual  $c_f(u) = f(u)$  —es decir, un borde que puede admitir flujo en la dirección opuesta a  $f(u)$ , como máximo cancelando el flujo en  $f(u)$ . Estos bordes inversos en la red residual permiten que un algoritmo envíe flujo de retorno

ya ha enviado a lo largo de un borde. Enviar flujo de regreso a lo largo de un borde es equivalente a disminuir el flujo en el borde, lo cual es una operación necesaria en muchos algoritmos.

Más formalmente, supongamos que tenemos una red de flujo  $G = (V, E)$  con fuente  $s$  y sumidero  $t$ . Sea  $f$  un flujo en  $G$ , y consideremos un par de vértices  $u, v \in V$ . Definimos la capacidad residual  $c_f(u, v)$  por

$$\begin{aligned} c_f(u, v) &= \begin{cases} 2 & \text{si } u \neq v \\ 0 & \text{de lo contrario} \end{cases} \quad (26.2) \end{aligned}$$

Debido a nuestra suposición de que  $c_f(u, v) \geq 0$  implica  $u \neq v$ , exactamente un caso en la ecuación (26.2) se aplica a cada par ordenado de vértices.

Como ejemplo de la ecuación (26.2), si  $c_f(s, v) = 16$  y  $c_f(v, t) = 11$ , entonces podemos aumentar  $c_f(s, v)$  hasta  $c_f(s, v) = 5$  unidades antes de que excedamos la restricción de capacidad en el borde  $v, t$ . También deseamos permitir que un algoritmo devuelva hasta 11 unidades de flujo de  $s$  a  $t$ , y por lo tanto  $c_f(s, t) = 11$ .

Dada una red de flujo  $G = (V, E)$  y un flujo  $f$ , la red residual de  $G$  inducida por  $f$  es  $G_f = (V, E_f)$ , donde

$$E_f = \{(u, v) \in E \mid c_f(u, v) > 0\} \quad (26.3)$$

Es decir, como se prometió anteriormente, cada borde de la red residual, o borde residual, puede admitir un flujo mayor que 0. La Figura 26.4(a) repite la red de flujo  $G$  y el flujo  $f$  de la Figura 26.1(b), y la Figura 26.4(b) muestra la red residual correspondiente  $G_f$ . Las aristas en  $E_f$  son aristas en  $E$  o sus inversiones, y por lo tanto

$$E_f = \{(u, v) \in E \mid c_f(u, v) > 0\}$$

Observe que la red residual  $G_f$  es similar a una red de flujo con capacidades dadas por  $c_f$ . No satisface nuestra definición de una red de flujo porque puede contener un borde  $u, v$  y su inversión  $v, u$ . Aparte de esta diferencia, una red residual tiene las mismas propiedades que una red de flujo, y podemos definir un flujo en la red residual como aquel que satisface la definición de flujo, pero con respecto a las capacidades  $c_f$  en la red  $G_f$ .

Un flujo en una red residual proporciona una hoja de ruta para agregar flujo al original red de flujo. Si  $f$  es un flujo en  $G$  y  $f'$  es un flujo en el residual correspondiente, definimos  $f'' = f + f'$  el aumento del flujo  $f$  por  $f'$  de  $V$  a  $R$ , definido por

$$\begin{aligned} f''(u, v) &= \begin{cases} c_f(u, v) & \text{si } (u, v) \in E_f \\ 0 & \text{de lo contrario} \end{cases} \quad (26.4) \end{aligned}$$

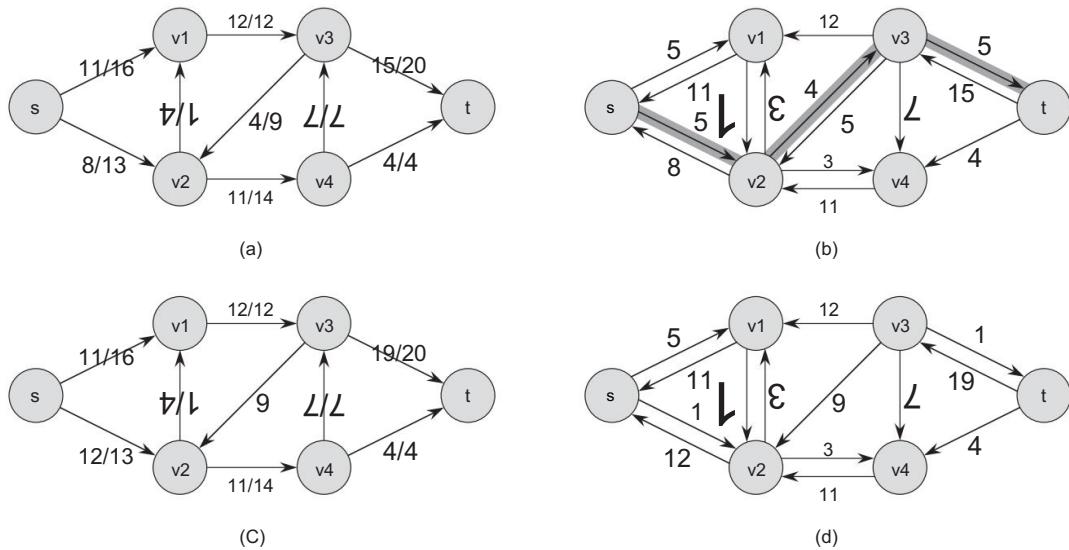


Figura 26.4 (a) La red de flujo  $G$  y el flujo  $f$  de la Figura 26.1 (b). (b) La red residual  $G_f$  con camino de aumento  $p$  sombreado; su capacidad residual es  $c_f(p) / D(c_f(p))$ . Bordes con capacidad residual igual a 0, como  $.1; .3/$ , no se muestran, una convención que seguimos en el resto de esta sección. (c) El flujo en  $G$  que resulta de aumentar a lo largo del camino  $p$  por su capacidad residual 4. Bordes que no llevan flujo, como  $.3/$ ;  $.2/$ , están etiquetados solo por su capacidad, otra convención que seguimos en todo momento. (d) La red residual inducida por el flujo en (c).

La intuición detrás de esta definición sigue la definición de la red residual. ; tu/ porque Aumentamos el flujo en  $.u/$ ; al empujar el  $^0.u/$  pero disminuyéndolo en  $f^0$  flujo en el borde inverso en la red residual significa disminuir el flujo en la red original. Empujar el flujo en el borde inverso de la red residual también se conoce como cancelación. Por ejemplo, si enviamos 5 cajas de discos de hockey de  $u$  a  $y$  enviamos 2 cajas de  $u$  a  $u$ , podríamos equivalentemente (desde la perspectiva del resultado final) enviar 3 creaciones de  $u$  a  $y$  ninguna de  $u$  a  $u$ .

La cancelación de este tipo es crucial para cualquier algoritmo de flujo máximo.

Lema 26.1 Sea

GD .V; Sea  $E$  una red de flujo con fuente  $s$  y sumidero  $t$ , y sea  $f$  un flujo en  $G$ . Sea  $G_f$  la red residual de  $G$  inducida por  $f$ , y sea  $f'$  un flujo en  $G_f$ . Entonces la función  $f'' = f + f'$  definida en la ecuación (26.4) es un flujo en  $G$  con valor  $j(f'')$

$$j(f'') = j(f) + j(f').$$

Prueba Primero verificamos que  $f''$  y  $j(f'')$  obedecen a la restricción de capacidad para cada borde en  $E$  la conservación del flujo en cada vértice en  $V \setminus \{s, t\}$ .

Para la restricción de capacidad, primero observe que si  $u; / 2 E$ , luego  $cf.; u; / D f .u; /$ . Por lo tanto, tenemos  $f.; u; / cf.; u; / D f .u; /$ , y por lo tanto

$$f'' f^0 / .u; / D f .u; / C f f .u; / ^0 .u; / f^0 ; u / \text{ (por la ecuación (26.4))}$$

$$C f D f^0 .u; / f .u; / \text{ (porque } f^0 ; u / f .u; /)$$

$$^0 .u; /$$

$$0$$

Además,  $f$

$$f'' f^0 / .u; /$$

$$D f .u; / C^0 .u; / f^0 ; u / \text{ (por la ecuación (26.4))}$$

$$ff .u; / C^0 .u; / \text{ (porque los flujos no son negativos)}$$

$$ff .u; / C f f .u; / \text{ (restricción de}$$

$$D f .u; / C cu; / f .u; / Dcu; / : \text{ capacidad) (definición de } cf \text{ )}$$

Para la

conservación del flujo, porque tanto  $f^0$  obedecer la conservación del flujo, tenemos como  $f$  que para todo  $u \in V$   $f(u) = \sum_{v \in N^+(u)} f(v) - \sum_{v \in N^-(u)}$

$$\sum_{v \in N^+(u)} f(v) - \sum_{v \in N^-(u)} f(v) = 0 \quad \forall u \in V$$

$$\sum_{v \in N^+(u)} f(v) - \sum_{v \in N^-(u)} f(v) = 0 \quad \forall u \in V$$

$$\sum_{v \in N^+(u)} f(v) - \sum_{v \in N^-(u)} f(v) = 0 \quad \forall u \in V$$

$$\sum_{v \in N^+(u)} f(v) - \sum_{v \in N^-(u)} f(v) = 0 \quad \forall u \in V$$

$$\sum_{v \in N^+(u)} f(v) - \sum_{v \in N^-(u)} f(v) = 0 \quad \forall u \in V$$

donde la tercera línea se sigue de la segunda por conservación del flujo.

Finalmente, calculamos el valor de  $f'' f^0$ . Recuerde que no permitimos aristas en  $G$  (pero no en  $G_f$ ), y por tanto para cada vértice  $v$ , antiparalelo sabemos que hay  $V$  puede ser una arista  $s; / o ; s/$ , pero nunca ambos. Definimos  $V_1 \subseteq \{v \in V \mid \text{existe } s; / v\}$  y  $V_2 \subseteq \{v \in V \mid \text{existe } v; / s\}$ . Ej. ser el conjunto de vértices con aristas de  $s$ , y  $V_2 \subseteq \{v \in V \mid \text{existe } v; / s\}$ . Ej. ser el conjunto de vértices con aristas a  $s$ . Tenemos  $|V_1| = |V_2|$  y, debido a que no permitimos aristas antiparalelas,  $|V_1 \setminus V_2| = |V_2 \setminus V_1|$ . ahora calculamos

$$\sum_{v \in V_1} f(v) - \sum_{v \in V_2} f(v) = \sum_{v \in V_1} f(v) - \sum_{v \in V_2} f(v) = 0$$

$$\sum_{v \in V_1} f(v) - \sum_{v \in V_2} f(v) = \sum_{v \in V_1} f(v) - \sum_{v \in V_2} f(v) = 0 \quad (26.5)$$

donde sigue la segunda línea porque  $f'' f^0 / w; x \neq 0$  si  $w; x \neq 62$ . Pasamos ahora a aplicar la definición de  $f'' f^0$  la ecuación (26.5), y luego reordenamos y agrupamos los términos para obtener

$$\begin{aligned}
 & jf'' f^0 \\
 & DX f.s; / Cf^0 .s; / f^0 .; s // X f .; s / C f^0 .; s / f^0 .s; // \\
 & \quad 2V1 \quad \quad \quad 2V2 \\
 & DX f.s; / CX f^0 .s; / Xf^0 .; s / \\
 & \quad 2V1 \quad \quad \quad 2V1 \\
 & \quad Xf.; s/Xf^0 .; s / CX f^0 .s; / \\
 & \quad 2V2 \quad \quad \quad 2V2 \\
 & DX f.s; / X f.; s / \\
 & \quad 2V1 \quad \quad \quad 2V2 \\
 & CX F^0 .s; / CX F^0 .s; / X F^0 .; s / X F^0 .; s / \\
 & \quad 2V1 \quad \quad \quad 2V2 \quad \quad \quad 2V1 \quad \quad \quad 2V2 \\
 & DX f.s; / Xf.; s / CX F^0 .s; / X F^0 .; s / : \quad (26.6)
 \end{aligned}$$

En la ecuación (26.6), podemos extender las cuatro sumas para sumar sobre  $V$ , desde cada uno término adicional tiene valor 0. (El ejercicio 26.2-1 le pide que demuestre esto formalmente). Así tenemos

$$\begin{aligned}
 & jf'' f^0 j DX f.s; / Xf.; s / CX f^0 .s; / X f^0 .; s / \\
 & \quad 2V \quad \quad \quad 2V \quad \quad \quad 2V \quad \quad \quad 2V \quad \quad \quad (26.7) \\
 & D jf j C jf^0 .; s /
 \end{aligned}$$

■

### Aumento de caminos

Dada una red de flujo  $G(V; E)$  y un flujo  $f$ , un camino creciente  $p$  es un camino simple de  $s$  a  $t$  en la red residual  $G_f$ . Por la definición de la red residual, podemos aumentar el flujo en un borde  $u; v$  de un camino que aumenta hasta  $c_f(u; v) /$  sin violar la restricción de capacidad en cualquiera de  $u; v$ ;  $u; v$  está en la red de flujo original  $G$ .

El camino sombreado en la Figura 26.4(b) es un camino de aumento. Si tratamos la red residual  $G_f$  de la figura como una red de flujo, podemos aumentar el flujo a través de cada borde de este camino hasta en 4 unidades sin violar una restricción de capacidad, ya que la capacidad residual más pequeña en este camino es  $c_f(2; 3) / 4$ . A la cantidad máxima por la que podemos aumentar el caudal en cada arista en un camino creciente  $p$  la llamamos capacidad residual de  $p$ , dada por

$c_f(p) = \min_{u; v \in p} c_f(u; v) / W(u; v)$  está en la página:

El siguiente lema, cuya demostración dejamos como ejercicio 26.2-7, hace más preciso el argumento anterior.

### Lema 26.2

Sea  $G \in V$ ; Sea  $E$  una red de flujo, sea  $f$  un flujo en  $G$ , y sea  $p$  un camino creciente en  $G_f$ . Defina una función  $f_p : E \rightarrow \mathbb{R}$  por

$$f_p(u; p) = \begin{cases} 0 & \text{si } u \in p \\ \min_{v \in p} f(v) & \text{de lo contrario} \end{cases} \quad (26.8)$$

Entonces,  $f_p$  es un flujo en  $G_f$  con valor  $\sum_{u \in p} f_p(u; p) > 0$ . ■

El siguiente corolario muestra que si aumentamos  $f$  por  $f_p$ , obtenemos otro flujo en  $G$  cuyo valor está más cerca del máximo. La figura 26.4(c) muestra el resultado de aumentar el flujo  $f$  de la figura 26.4(a) por el flujo  $f_p$  de la figura 26.4(b), y la figura 26.4(d) muestra la red residual resultante.

### Corolario 26.3

Sea  $G \in V$ ; Sea  $E$  una red de flujo, sea  $f$  un flujo en  $G$ , y sea  $p$  un camino creciente en  $G_f$ . Sea  $f_p$  definida como en la ecuación (26.8), y suponga que aumentamos  $f$  por  $f_p$ . Entonces la función  $f + f_p$  es un flujo en  $G$  con valor  $\sum_{u \in p} (f(u) + f_p(u; p)) > \sum_{u \in p} f(u)$ .

Prueba inmediata de los lemas 26.1 y 26.2. ■

### Cortes de redes de flujo

El método Ford-Fulkerson aumenta repetidamente el flujo a lo largo de caminos crecientes hasta que encuentra un flujo máximo. ¿Cómo sabemos que cuando el algoritmo termina, en realidad hemos encontrado un flujo máximo? El teorema max-flow min-cut, que demostrarímos en breve, nos dice que un flujo es máximo si y solo si su red residual no contiene una ruta de aumento. Sin embargo, para probar este teorema, primero debemos explorar la noción de corte de una red de flujo.

Un corte  $(S, T)$  de caudal  $G \in V$ ;  $E$  es una partición de  $V$  en  $S$  y  $T$  tal que  $s \in S$  y  $t \in T$ . (Esta definición es similar a la definición de “corte” que usamos para los árboles de expansión mínima en el Capítulo 23, excepto que aquí estamos cortando un grafo dirigido en lugar de un grafo no dirigido, e insistimos en que  $s \in S$  y  $t \in T$ .) Si  $f$  es un flujo, entonces el flujo neto  $f(S, T)$  a través del corte  $(S, T)$  se define como

$$\sum_{u \in S, v \in T} f(u, v) - \sum_{v \in T, u \in S} f(v, u) \quad (26.9)$$

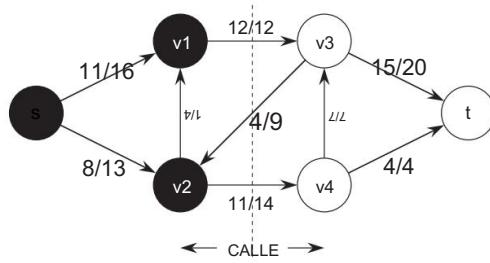


Figura 26.5 Un corte  $.S; T /$  en la red de flujo de la Figura 26.1(b), donde  $S = \{s, v_1, v_2\}$ ;  $T = \{v_3, v_4, t\}$ . Los vértices en  $S$  son negros y los vértices en  $T$  son blancos. El flujo neto a través de  $.S; T /$  es  $f(.S; T) = 19$ , y la capacidad es  $c(S; T) = 26$ .

La capacidad del corte  $.S; T /$  es

$$\frac{c(S; T)}{c(S; T)} = \frac{\sum_{e \in S \times T} c_e - \sum_{e \in T \times S} c_e}{\sum_{e \in S \times T} c_e + \sum_{e \in T \times S} c_e} : \quad (26.10)$$

Un corte mínimo de una red es un corte cuya capacidad es mínima sobre todos los cortes de la red.

La simetría entre las definiciones de flujo y capacidad de un corte es intencional e importante. Para la capacidad, contamos solo las capacidades de los bordes que van de  $S$  a  $T$ , ignorando los bordes en la dirección inversa. Para flujo, consideramos el flujo que va de  $S$  a  $T$  menos el flujo que va en la dirección inversa de  $T$  a  $S$ . La razón de esta diferencia se aclarará más adelante en esta sección.

La Figura 26.5 muestra el corte  $.S; T /$  en la red de flujo de la figura 26.1(b). El flujo neto a través de este corte es

$$f(.S; T) = f(v_1 \rightarrow v_3) - f(v_3 \rightarrow v_1) + f(v_2 \rightarrow v_3) - f(v_3 \rightarrow v_2) + f(v_2 \rightarrow v_4) - f(v_4 \rightarrow v_2) + f(v_4 \rightarrow t) - f(t \rightarrow v_4) = 19$$

$$= D 19$$

y la capacidad de este corte es

$$c(.S; T) = \min_{e \in S \times T} c_e = 14$$

El siguiente lema muestra que, para un flujo  $f$  dado, el flujo neto a través de cualquier corte es el mismo, y es igual a  $f(j)$ , el valor del caudal.

Lema 26.4

Sea  $f$  un flujo en una red de flujo  $G$  con fuente  $s$  y sumidero  $t$ , y sea  $.S; T /$  ser cualquier corte de  $G$ . Entonces el flujo neto a través de  $.S; T /$  es  $f(.S; T) = f(j)$ .

Prueba Podemos reescribir la condición de conservación de flujo para cualquier nodo u 2 V fs;tg  
como

$$\frac{X f .u;}{2V} / \frac{X f.;}{2V} U / D 0 \quad (26.11)$$

Tomando la definición de  $j_f j$  de la ecuación (26.1) y sumando el lado izquierdo de la ecuación (26.11), que es igual a 0, sumando todos los vértices en S fsg, se obtiene

$$\frac{DX}{2V} \frac{f.s;}{2V} / \frac{X f.;}{2V} s / CX j_f j \quad \frac{X f .u;}{2V} / \frac{X}{2V} f.; tu / ! :$$

Expandiendo la suma de la mano derecha y reagrupando términos se obtiene

$$\begin{aligned} & j_f j \frac{DX}{2V} \frac{f.s;}{2V} / \frac{X f.;}{2V} s / CX \quad \frac{X f .u;}{2V} / X \quad \frac{X}{2V} f.; tu / ! \\ & \frac{DX}{2V} \frac{f.s;}{2V} / CX \quad f .u; / ! X \quad f.; s / CX \quad f.; tu / ! \\ & DXX f .u; / XX s.; tu / : \end{aligned}$$

Como VDS [ T y S \ TD ;, podemos dividir cada sumatoria sobre V en sumatorias sobre S y T para obtener

$$\begin{aligned} & j_f j \frac{DXX}{2S} \frac{f.u;}{2T} / CX \quad X f .u; / XX \quad s.; u / XX f.; tu / \\ & DXX f .u; / XX s.; tu / \\ & CXX f .u; / XX \quad f.; tu / ! : \end{aligned}$$

Las dos sumatorias entre paréntesis son en realidad las mismas, ya que para todos los vértices x; y 2 V el término  $f .x; y/$  aparece una vez en cada sumatoria. Por lo tanto, estas sumatorias se cancelan y tenemos

$$\begin{aligned} & j_f j \frac{DXX}{u2S} \frac{f.u;}{2T} / XX \quad f.; tu / \\ & D f .S; T / : \end{aligned}$$

■

Un corolario del Lema 26.4 muestra cómo podemos usar las capacidades de corte para limitar el valor de un flujo.

Corolario 26.5 El

valor de cualquier flujo  $f$  en una red de flujo  $G$  está acotado superiormente por la capacidad de cualquier corte de  $G$ .

Prueba Sea  $.S$ ; Sea  $T$  / cualquier corte de  $G$  y sea  $f$  cualquier flujo. Por el Lema 26.4 y la restricción de

$$\begin{aligned} & \text{capacidad, } jf \leq D_f .S; T / D_{XX} f .u; / XX s.; tu / \\ & \quad u2S \quad 2T \quad u2S \quad 2T \\ & X \quad X \quad f .u; / \\ & \quad u2S \quad 2T \\ & XX \text{ pies cúbicos; } / \\ & \quad u2S \quad 2T \\ & DcS; T : \end{aligned}$$

■

El corolario 26.5 produce la consecuencia inmediata de que el valor de un caudal máximo en una red está acotado superiormente por la capacidad de un corte mínimo de la red. El importante teorema de corte mínimo de flujo máximo, que ahora enunciamos y probamos, dice que el valor de un flujo máximo es de hecho igual a la capacidad de un corte mínimo.

Teorema 26.6 (Teorema de corte mínimo de flujo máximo)

Si  $f$  es un flujo en una red de flujo  $G$  .V; E/ con fuente  $s$  y sumidero  $t$ , entonces las siguientes condiciones son

- equivalentes: 1.  $f$  es un caudal máximo en  $G$ .
- 2. La red residual  $G_f$  no contiene caminos de aumento.
- 3.  $jf \geq D_{cS; T}$  por algún corte  $.S; T$  de  $G$ .

Demostración .1/ ) .2/: Supongamos, en aras de la contradicción, que  $f$  es un flujo máximo en  $G$  pero que  $G_f$  tiene un camino creciente  $p$ . Entonces, por el Corolario 26.3, el flujo encontrado al aumentar  $f$  por  $f_p$ , donde  $f_p$  viene dado por la ecuación (26.8), es un flujo en  $G$  con un valor estrictamente mayor que  $jf$ , contradiciendo la suposición de que  $f$  es un flujo máximo.

.2/ ) .3/: Supongamos que  $G_f$  no tiene camino de aumento, es decir, que  $G_f$  contiene no hay camino de  $s$  a  $t$ .

Definir SD  $f$  2 VW existe un camino de  $s$  a en  $G_f$

$g$  y TDV  $S$ . La partición  $.S; T$  / es un corte: tenemos  $s$  2  $S$  trivialmente y  $t$  62  $S$  porque no hay camino de  $s$  a  $t$  en  $G_f$ . Ahora considere un par de vértices

u 2 S y 2 T Si .u; / 2 E, debemos tener f .u; / Dcu; /, ya que de lo contrario .u; / 2 Ef , que lo colocaría en el conjunto S. Si .u; / 2 E, debemos tener f .u; / D 0, porque de lo contrario cf .u; / D f .u; / u/ sería positivo y tendríamos .u; / 2 Ef , que lo colocaría en S. Por supuesto, si ni .u; / ni .u; / u/ está en E, entonces f .u; / D f .u; / u/ D 0. Así tenemos f .S; T/DXX f .u; / XX s.; tu/

$$\begin{array}{ccc}
 u2S & 2T & 2T u2S \\
 DXX cu; /XX & & 0 \\
 u2S & 2T & 2T u2S \\
 DcS; T / : & &
 \end{array}$$

Por el Lema 26.4, por lo tanto, jf j D f .S; T/D cS; t/.

.3/ ) .1/: Por el Corolario 26.5, jf j cS; T/ para todos los cortes .S; t/. La condición jf j D cS; T / por lo tanto implica que f es un caudal máximo. ■

#### El algoritmo básico de Ford-Fulkerson

En cada iteración del método de Ford-Fulkerson, encontramos alguna ruta de aumento p y usamos p para modificar el flujo f. Como sugieren el Lema 26.2 y el Corolario 26.3, reemplazamos f por f " fp, obteniendo un nuevo flujo cuyo valor es jf j C jfpj. La siguiente implementación del método calcula el flujo máximo en una red de flujo

GD .V; E/ actualizando el atributo de flujo .u; /:f para cada arista .u; / 2 e. 1

Si tu; / 62 E, asumimos implícitamente que .u; /:f D 0. También suponemos que tenemos las capacidades cu; / junto con la red de flujo, y cu; / D 0 si .u; / 62 E. Calculamos la capacidad residual cf .u; / de acuerdo con la fórmula (26.2). La expresión cf .p/ en el código es solo una variable temporal que almacena la capacidad residual de la ruta p.

#### FORD-FULKERSON.G; s; t/ 1

por cada arista .u; / 2 G:E 2 .u; /:f

D 0 3 mientras existe

un camino p de sa t en la red residual Gf 4 cf .p/ D min fcf .u; / W .u; / está

en pg para cada arista .u; / en p si .u; / 2 UE .u; //f

D .u; //f C cf .p/ si no .u; / u:/f

5 6 D ; u:/f cf .p/

7

8

---

1Recuerde de la Sección 22.1 que representamos un atributo f para la arista .u; / con el mismo estilo de notación—.u; /:f—que usamos para un atributo de cualquier otro objeto.

El algoritmo FORD-FULKERSON simplemente amplía el pseudocódigo del MÉTODO FORD-FULKERSON dado anteriormente. La figura 26.6 muestra el resultado de cada iteración en una ejecución de muestra. Las líneas 1 y 2 inicializan el flujo  $f$  a 0. El ciclo while de las líneas 3 a 8 encuentra repetidamente una trayectoria creciente  $p$  en  $G_f$  y aumenta el flujo  $f$  a lo largo de  $p$  por la capacidad residual  $c_f(p)$ . Cada borde residual en la ruta  $p$  es un borde en la red original o la inversión de un borde en la red original. Las líneas 6 a 8 actualizan el flujo en cada caso de manera apropiada, agregando flujo cuando el borde residual es un borde original y restándolo en caso contrario. Cuando no existen caminos de aumento, el flujo  $f$  es un flujo máximo.

### Análisis de Ford-Fulkerson

El tiempo de ejecución de FORD-FULKERSON depende de cómo encontramos el camino de aumento  $p$  en la línea 3. Si lo elegimos mal, es posible que el algoritmo ni siquiera termine: el valor del flujo aumentará con aumentos sucesivos, pero ni siquiera necesita converger al valor de flujo máximo.<sup>2</sup> Sin embargo, si encontramos la ruta de aumento usando una búsqueda en anchura (que vimos en la Sección 22.2), el algoritmo se ejecuta en tiempo polinomial. Antes de probar este resultado, obtenemos un límite simple para el caso en el que elegimos arbitrariamente el camino de aumento y todas las capacidades son números enteros.

En la práctica, el problema del flujo máximo a menudo surge con capacidades integrales. Si las capacidades son números racionales, podemos aplicar una transformación de escala apropiada para hacerlos todos enteros. Si  $f$  denota un flujo máximo en la red transformada, entonces una implementación sencilla de FORD-FULKERSON ejecuta el bucle while de las líneas 3 a 8 como máximo  $jf$  veces, ya que el valor del flujo aumenta al menos una unidad en cada iteración.

Podemos realizar el trabajo realizado dentro del ciclo while de manera eficiente si implementamos la red de flujo  $G$  . $V$ ;  $E$  con la estructura de datos correcta y encuentre una ruta de aumento mediante un algoritmo de tiempo lineal. Supongamos que mantenemos una estructura de datos correspondiente a un grafo dirigido  $G_0$  . $V$ ;  $E_0$  /, donde  $E_0$  D  $f_u$ ; /  $W$  . $u$ ; / 2  $E$  o ;  $u$  / 2  $E$ . Los bordes en la red  $G$  también son bordes , y por lo tanto podemos en  $G_0$  y mantienen fácilmente las capacidades y los flujos en esta estructura de datos. Dado un flujo  $f$  en  $G$ , las aristas en la red residual  $G_f$  consisten en todas las aristas . $u$ ; / de  $G_0$  tal que  $c_f(u) > 0$ , donde  $c_f$  se ajusta a la ecuación (26.2). El tiempo para encontrar un camino en una red residual es, por lo tanto,  $O(V^2 E_0)$  /  $O(E)$  si usamos la búsqueda primero en profundidad o la búsqueda primero en amplitud. Por lo tanto, cada iteración del ciclo while toma tiempo  $O(E)$ , al igual que la inicialización en las líneas 1 y 2, lo que hace que el tiempo total de ejecución del algoritmo FORD-FULKERSON sea  $O(E j f)$ .

---

<sup>2</sup>El método de Ford-Fulkerson podría fallar en la terminación solo si las capacidades de borde son números irracionales.

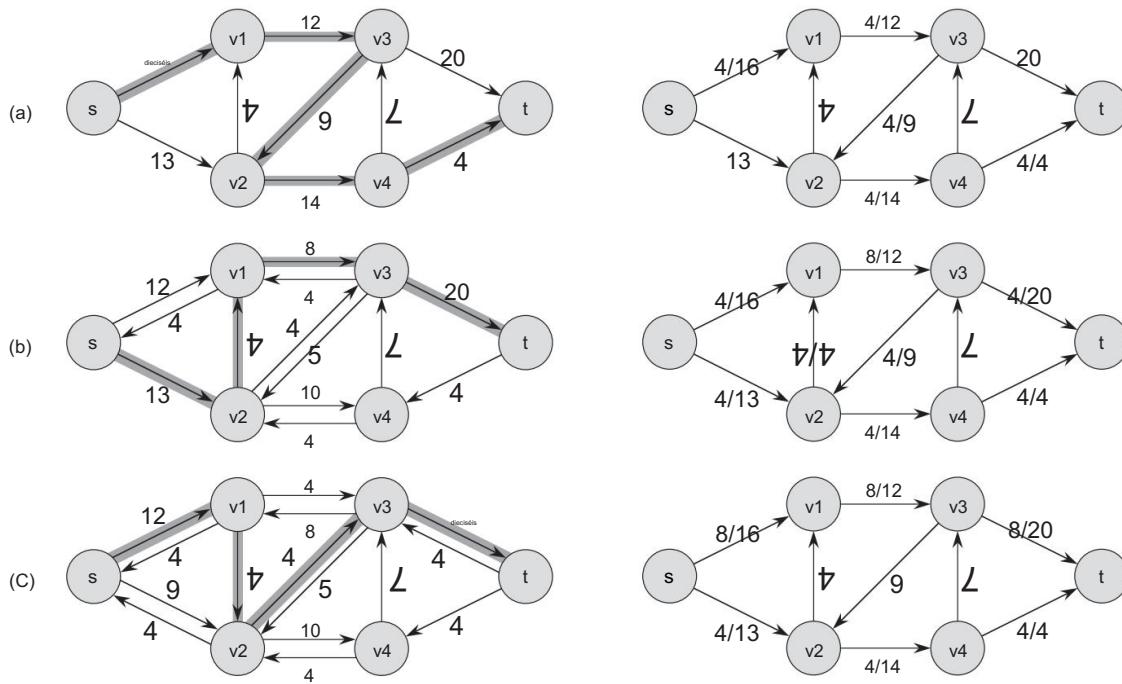


Figura 26.6 La ejecución del algoritmo básico de Ford-Fulkerson. (a)–(e) Iteraciones sucesivas del ciclo while . El lado izquierdo de cada parte muestra la red residual  $G_f$  de la línea 3 con un camino creciente sombreado  $p$ . El lado derecho de cada parte muestra el nuevo flujo  $f$  que resulta de aumentar  $f$  por  $f_p$ . La red residual en (a) es la red de entrada  $G$ .

Cuando las capacidades son integrales y el valor de flujo óptimo  $j_f$  es pequeño, el tiempo de ejecución del algoritmo de Ford-Fulkerson es bueno. La figura 26.7(a) muestra un ejemplo de lo que puede suceder en una red de flujo simple para la cual  $j_f$  es grande. Un caudal máximo en esta red tiene un valor de 2.000.000: ¡1.000.000 de unidades de caudal recorren el camino  $s \rightarrow t$ , y otras 1.000.000 unidades recorren el camino  $s \rightarrow t$ . Si el primer camino de aumento encontrado por FORD-FULKERSON es  $s \rightarrow t$ , que se muestra en la figura 26.7(a), el flujo tiene valor 1 después de la primera iteración. La red residual resultante aparece en la figura 26.7(b). Si la segunda iteración encuentra la ruta de aumento  $s \rightarrow t$ , como se muestra en la Figura 26.7(b), el flujo entonces tiene valor 2. La figura 26.7(c) muestra la red residual resultante. ¡Podemos continuar, eligiendo los caminos de aumento  $s \rightarrow t$  en las iteraciones impares y las rutas de aumento  $s \rightarrow t$  en las iteraciones pares. Realizaríamos un total de 2.000.000 de aumentos, aumentando el valor de flujo en solo 1 unidad en cada uno.

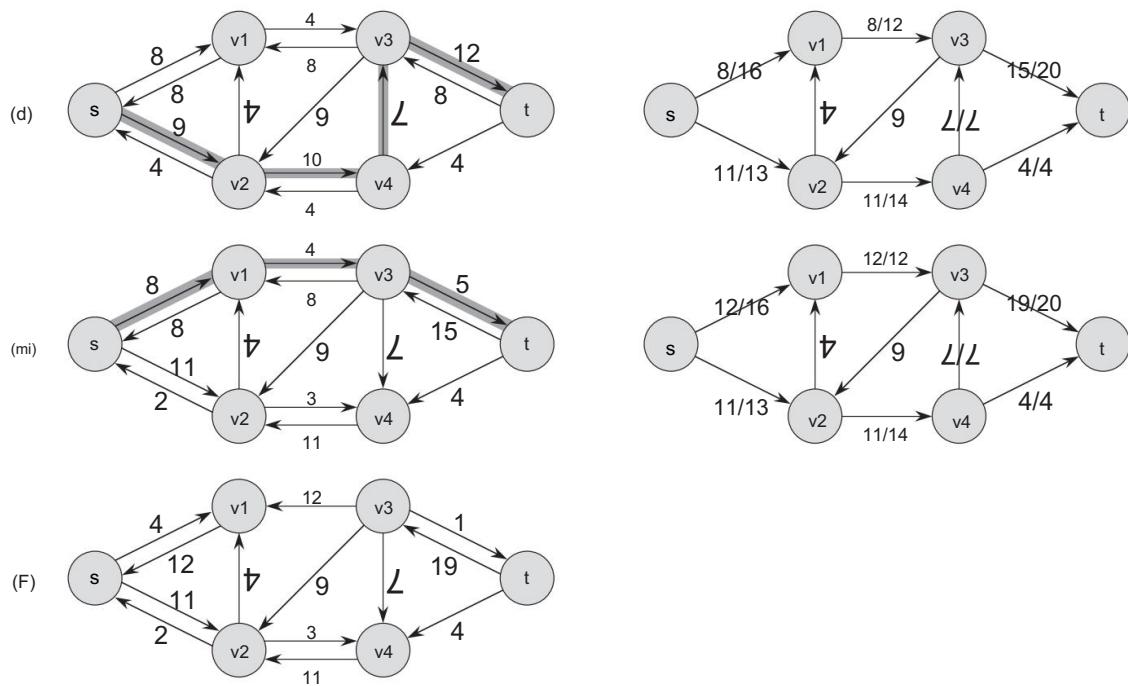


Figura 26.6, continuación (f) La red residual en la última prueba de bucle while . No tiene trayectorias crecientes y, por lo tanto, el flujo  $f$  que se muestra en (e) es un flujo máximo. El valor del caudal máximo encontrado es 23.

### El algoritmo de Edmonds-Karp

Podemos mejorar el límite de FORD-FULKERSON encontrando la ruta de aumento  $p$  en la línea 3 con una búsqueda en anchura. Es decir, elegimos la ruta de aumento como la ruta más corta de  $s$  a  $t$  en la red residual, donde cada borde tiene una unidad de distancia (peso). Llamamos al método de Ford-Fulkerson así implementado el algoritmo de Edmonds-Karp. Ahora demostramos que el algoritmo de Edmonds-Karp se ejecuta en tiempo  $O(V E^2)$ .

El análisis depende de las distancias a los vértices en la red residual  $G_f$ . El siguiente lema usa la notación  $\text{if } u; /$  para la distancia del camino más corto desde  $u$  hasta  $v$  en  $G_f$ , donde cada borde tiene una unidad de distancia.

Lema 26.7 Si

el algoritmo de Edmonds-Karp se ejecuta en una red de flujo GD .V; E/ con fuente s y sumidero t, entonces para todos los vértices  $2 V_{fs}; t_g$ , la distancia del camino más corto  $\text{if } s; /$  en la red residual  $G_f$  aumenta monótonamente con cada aumento de flujo.

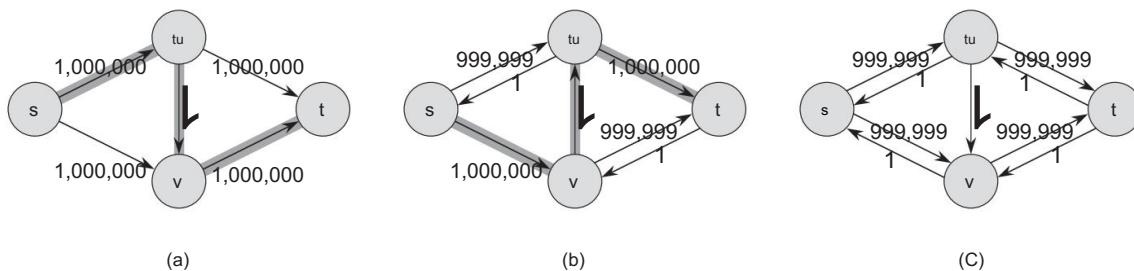


Figura 26.7 (a) Una red de flujo para la cual FORD-FULKERSON puede tomar „E jf j/ tiempo, donde f es un flujo máximo, que se muestra aquí con jf j D 2,000,000. El camino sombreado es un camino de aumento con capacidad residual 1. (b) La red residual resultante, con otro camino de aumento cuya capacidad residual es 1. (c) La red residual resultante.

Demostración Supondremos que para algún vértice  $2 \leq i \leq t$ , hay un aumento del flujo que hace que la distancia del camino más corto de  $s$  a  $i$  disminuya, y entonces deduciremos una contradicción. Sea  $f$  el flujo justo antes del primer aumento que disminuye alguna distancia del camino más corto, y sea  $f'$  el flujo justo después.

Sea el vértice con el mínimo  $d(s; i)$  cuya distancia fue disminuida por el aumento, de modo que  $d(s; i) < d(s; i')$ . Sea  $p$  el camino más corto de  $s$  a  $i$  en  $G_f$ , de modo que  $d(s; i) = |p|$  y  $d(s; i') \geq |p| + 1$ .

$$(26.12) \quad 1$$

Debido a cómo elegimos no  $i$ , sabemos que la distancia del vértice  $i$  a la fuente  $s$  disminuyó, es decir,

$$(26.13) \quad \text{si } d(s; i) < d(s; i') \text{ :}$$

Decimos que  $d(s; i) \leq 2d(s; p)$ . ¿Por qué? Si tuviéramos  $d(s; i) \geq 3d(s; p)$ , entonces también tendríamos  $d(s; i') \geq 3d(s; p)$ .

Sea  $p$  el camino más corto de  $s$  a  $i$  en  $G_f$ , de modo que  $d(s; i) = |p|$  y  $d(s; i') \geq |p| + 1$  (por el Lema 24.10, la desigualdad triangular)  $d(s; i) \leq 2d(s; p)$  (por la desigualdad (26.13))

$d(s; i) \leq 2d(s; p)$  (por la ecuación (26.12)),

lo que contradice nuestra suposición de que  $d(s; i) < d(s; i')$ .

¿Cómo podemos tener  $d(s; i) \leq 2d(s; p)$ ? El aumento debe haber aumentado el flujo de  $s$  a  $i$ . El algoritmo de Edmonds-Karp siempre aumenta el flujo a lo largo de los caminos más cortos y, por lo tanto, el camino más corto de  $s$  a  $i$  en  $G_f$  tiene  $i$  como su última arista. Por lo tanto, si  $d(s; i) \leq 2d(s; p)$ ,  $i$  es el vértice  $t$  (por la desigualdad (26.13))

$d(s; i) \leq 2d(s; p)$  (por la ecuación (26.12)),

lo que contradice nuestra suposición de que  $\text{if } 0.s; / < \text{ si } .s; /$ . Concluimos que nuestra suposición de que tal vértice existe es incorrecta. ■

El siguiente teorema limita el número de iteraciones del algoritmo de Edmonds-Karp.

#### Teorema 26.8

Si el algoritmo de Edmonds-Karp se ejecuta en una red de flujo  $G = (V, E)$  con fuente  $s$  y sumidero  $t$ , entonces el número total de aumentos de flujo realizados por el algoritmo es  $O(|V|E)$ .

Prueba Decimos que una arista  $(u, v)$  en una red residual  $G_f$  es crítica en un trayecto de aumento  $p$  si la capacidad residual de  $p$  es la capacidad residual de  $(u, v)$ , es decir, si  $c_f(p) = c_f((u, v))$ . Una vez que hemos aumentado el flujo a lo largo de una ruta de aumento, cualquier borde crítico en la ruta desaparece de la red residual. Además, al menos un borde en cualquier ruta de aumento debe ser crítico. Mostraremos que cada una de las aristas  $(v, u)$  puede volverse crítica como máximo  $|V| - 2$  veces.

Sean  $u$  y vértices en  $V$  que están conectados por una arista en  $E$ . Como aumentar los caminos son los caminos más cortos, cuando  $(u, v)$  es crítico por primera vez, tenemos

$\text{si } .s; / D \text{ if } .s; tu/C 1$

Una vez que se aumenta el flujo, el borde  $(u, v)$  desaparece de la red residual.

No puede reaparecer más tarde en otro camino de aumento hasta después de que el flujo de  $u$  a disminuya, lo que ocurre solo si  $(v, u)$  aparece en un camino de aumento. Si  $f$  es el flujo en  $G^0$  cuando ocurre este evento, entonces tenemos

$\text{si } 0.s; u / D \text{ si } 0.s; / C 1$

Dado que  $\text{si } .s; / \text{ si } 0.s; /$  por el Lema 26.7, tenemos  $\text{si } 0.s;$

$u / D \text{ si } 0.s; / C 1 \text{ si } .s; / C 1 D$

$\text{if } .s; tu/C 2$

En consecuencia, desde el tiempo  $(u, v)$  se vuelve crítico para el momento en que luego se vuelve crítico, la distancia de  $v$  desde la fuente aumenta en al menos 2. La distancia de  $v$  desde la fuente es inicialmente al menos 0. Los vértices intermedios en un camino más corto de  $s$  a  $v$  no pueden contener  $s$ ,  $u$ , o  $t$  (ya que  $(u, v)$  en un camino de aumento implica que  $u \neq t$ ). Por lo tanto, hasta que  $v$  se vuelve inalcanzable desde la fuente, si es que alguna vez ocurre, su distancia es como mucho  $|V| - 2$ . Así, después de la primera vez que  $(u, v)$  se vuelve crítico, puede volverse crítico como máximo  $|V| - 2 = |V| - 2$  veces más, para un total de como máximo  $|V| - 2$  veces. Dado que hay  $|V|$  pares de vértices que pueden tener una arista entre ellos en una red residual, el número total de aristas críticas durante

la ejecución completa del algoritmo de Edmonds-Karp es  $O(VE)$ . Cada camino de aumento tiene al menos un borde crítico y, por lo tanto, se sigue el teorema. ■

Debido a que podemos implementar cada iteración de FORD-FULKERSON en tiempo  $O(VE)$  cuando encontramos la ruta de aumento mediante la búsqueda en amplitud, el tiempo total de ejecución del algoritmo de Edmonds-Karp es  $O(VE^2)$ . Veremos que los algoritmos push-relabel pueden producir límites aún mejores. El algoritmo de la Sección 26.4 proporciona un método para lograr un tiempo de ejecución  $O(V^2E)$ , que constituye la base del algoritmo  $O(V^3/\text{tiempo})$  de la Sección 26.5.

### Ejercicios

#### 26.2-1

Demuestre que las sumatorias de la ecuación (26.6) son iguales a las sumatorias de la ecuación (26.7).

#### 26.2-2

En la figura 26.1(b), ¿cuál es el flujo a través del corte  $\{s; 2; 4g; f1; 3; tg\}$ ? ¿Cuál es la capacidad de este corte?

#### 26.2-3

Muestre la ejecución del algoritmo de Edmonds-Karp en la red de flujo de la figura 26.1(a).

#### 26.2-4

En el ejemplo de la figura 26.6, ¿cuál es el corte mínimo correspondiente al caudal máximo mostrado? De los caminos de aumento que aparecen en el ejemplo, ¿cuál cancela el flujo?

#### 26.2-5

Recuerde que la construcción en la Sección 26.1 que convierte una red de flujo con múltiples fuentes y sumideros en una red de una sola fuente y un solo sumidero agrega bordes con capacidad infinita. Demuestre que cualquier flujo en la red resultante tiene un valor finito si los bordes de la red original con múltiples fuentes y sumideros tienen una capacidad finita.

#### 26.2-6

Suponga que cada fuente  $s_i$  en una red de flujo con múltiples fuentes y sumideros produce exactamente  $p_i$  unidades de flujo, de modo que  $\sum_j f_{ij} = p_i$ . Supongamos también que cada sumidero  $t_j$  consume exactamente  $q_j$  unidades, de modo que  $\sum_i f_{ij} = q_j$ , donde  $f_{ij}$  es el flujo entre  $s_i$  y  $t_j$ . Muestre cómo convertir el problema de encontrar un flujo  $f$  que obedezca

estas restricciones adicionales en el problema de encontrar un flujo máximo en una red de flujo de fuente única y sumidero único.

26.2-7

Demostrar Lema 26.2.

26.2-8

Suponga que redefinimos la red residual para no permitir bordes en  $s$ . Argumente que el procedimiento FORD-FULKERSON todavía calcula correctamente un caudal máximo.

26.2-9

Suponga que tanto  $f$  como  $f^0$  son flujos en una red  $G$  y calculamos el flujo  $f'' = f - f^0$ . ¿El flujo aumentado satisface la propiedad de conservación del flujo? ¿Satisface la restricción de capacidad?

26.2-10

Muestre cómo encontrar un flujo máximo en una red  $G = (V, E)$  por una secuencia de caminos de aumento como máximo jEj. (Sugerencia: determine las rutas después de encontrar el flujo máximo).

26.2-11

La conectividad de aristas de un grafo no dirigido es el número mínimo  $k$  de aristas que deben eliminarse para desconectar el grafo. Por ejemplo, la conectividad de borde de un árbol es 1 y la conectividad de borde de una cadena cíclica de vértices es 2. Muestre cómo determinar la conectividad de borde de un grafo no dirigido  $G = (V, E)$  ejecutando un algoritmo de flujo máximo en redes de flujo jV j como máximo, cada una con vértices OV/ y aristas OE/.

26.2-12

Suponga que tiene una red de flujo  $G$  y  $G$  tiene aristas que ingresan a la fuente  $s$ . Sea  $f$  un flujo en  $G$  en el que una de las aristas  $s; v$  entrando en la fuente tiene  $f_s < D_{s,v}$ . Demostrar que debe existir otro flujo  $f' = f + f^*$  tal que  $f'_s = f_s + 1$  y suponiendo que todas las capacidades de borde son números enteros. Proporcione un algoritmo OE-tiempo para calcular  $f'$ , dado  $f$ ,

26.2-13

Suponga que desea encontrar, entre todos los cortes mínimos en una red de flujo  $G$  con capacidades integrales, uno que contenga el menor número de aristas. Muestre cómo modificar las capacidades de  $G$  para crear una nueva red de flujo  $G_0$  en la que cualquier corte mínimo en  $G_0$  es un corte mínimo con el menor número de aristas en  $G$ .

### 26.3 Coincidencia bipartita máxima

Algunos problemas combinatorios se pueden convertir fácilmente en problemas de flujo máximo. El problema de flujo máximo de múltiples fuentes y múltiples sumideros de la sección 26.1 nos dio un ejemplo. Algunos otros problemas combinatorios parecen tener poco que ver con las redes de flujo, pero de hecho pueden reducirse a problemas de flujo máximo.

Esta sección presenta uno de esos problemas: encontrar una coincidencia máxima en un gráfico bipartito. Para resolver este problema, aprovecharemos una propiedad de integralidad proporcionada por el método de Ford-Fulkerson. También veremos cómo usar el método de Ford-Fulkerson para resolver el problema de emparejamiento bipartito máximo en un gráfico GD .V; E/ en O.VE/ tiempo.

#### El problema de coincidencia bipartita máxima

Dado un grafo no dirigido GD .V; E/, una coincidencia es un subconjunto de aristas ME tal que para todos los vértices  $2 V$  como máximo una arista de M incide sobre . Decimos que un vértice  $2 V$  coincide con el correspondiente M si alguna arista en M incide sobre ; de lo contrario, es inigualable. Un emparejamiento máximo es un emparejamiento de máxima cardinalidad, es decir, un emparejamiento M tal que para cualquier emparejamiento M0 tenemos  $jM \geq jM_0$ . En esta sección, , limitaremos nuestra atención a encontrar coincidencias máximas en grafos bipartitos: grafos en los que el conjunto de vértices se puede dividir en  $VL \cup VR$ , donde  $L$  y  $R$  son disjuntos y todas las aristas en E van entre L y R. suponga que todo vértice en V tiene al menos una arista incidente. La figura 26.8 ilustra la noción de coincidencia en un gráfico bipartito.

El problema de encontrar una coincidencia máxima en un gráfico bipartito tiene muchas aplicaciones prácticas. Como ejemplo, podríamos considerar hacer coincidir un conjunto L de máquinas con un conjunto R de tareas que se realizarán simultáneamente. Tomamos la presencia de edge  $u; v$  en E para significar que una máquina en particular  $u$  en  $L$  es capaz de realizar una tarea en particular  $v$  en  $R$ . Una coincidencia máxima proporciona trabajo para tantas máquinas como sea posible.

#### Encontrar una coincidencia bipartita máxima

Podemos usar el método de Ford-Fulkerson para encontrar una coincidencia máxima en un grafo bipartito no dirigido GD .V; E/ en polinomio de tiempo en  $|V|$  y  $|E|$ . El truco consiste en construir una red de flujo en la que los flujos correspondan a coincidencias, como se muestra en la figura 26.8(c). Definimos la red de flujo correspondiente  $G_0 = (V, E_0)$  para el gráfico bipartito G de la siguiente manera. Dejamos que la fuente s y el receptor t sean nuevos vértices no en  $V$ . Si la parte de  $V$  de  $G$  es  $VL \cup VR$ , la

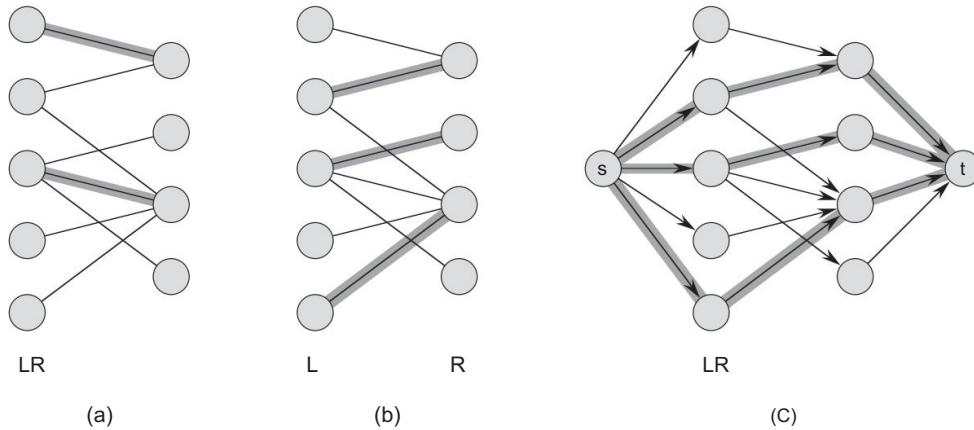


Figura 26.8 Un grafo bipartito  $G = (V; E)$  con partición de vértice  $V = L \cup R$ . (a) Una coincidencia con cardinalidad 2, indicada por bordes sombreados. (b) Una coincidencia máxima con cardinalidad 3. (c) La red de flujo correspondiente  $G_0$  con un flujo máximo mostrado. Cada borde tiene capacidad unitaria. Los bordes sombreados tienen un flujo de 1 y todos los demás bordes no tienen flujo. Los bordes sombreados de L a R corresponden a los de la coincidencia máxima de (b).

Las aristas dirigidas de  $G_0$  son las aristas de  $E$ , dirigidas de  $L$  a  $R$ , junto con  $jV - j$  nuevas aristas dirigidas:

$$E_0 = \{f_s : u \in W_u, u \in L \setminus f_u\} \cup \{f_u : u \in W_u, u \in R \setminus f_u\} \cup \{f_{u,v} : u \in W_u, v \in R, (u, v) \in E\}$$

Para completar la construcción, asignamos capacidad unitaria a cada borde en  $E_0$ . Como cada vértice en  $V$  tiene al menos una arista incidente,  $\sum_{v \in N(u)} \deg(v) = 2$ . Así,  $\sum_{v \in N(u)} \deg(v) = 2$  para todo  $u \in V$ .

El siguiente lema muestra que una coincidencia en  $G$  corresponde directamente a un flujo en la red de flujo  $G_0$  correspondiente a  $G$ . Decimos que un flujo  $f$  en una red de flujo  $G = (V; E)$  es de valor entero si  $f(u) \in \mathbb{Z}$  para todo  $u \in V$ .

#### Lema 26.9

Sea  $G = (V; E)$  un grafo bipartito con partición de vértices  $V = L \cup R$ , y sea  $G_0 = (V; E_0)$  su red de flujo correspondiente. Si  $M$  es una coincidencia en  $G$ , entonces hay un flujo  $f$  de valor entero en  $G_0$  con valor  $\sum_{(u, v) \in M} f(u, v)$ . Por el contrario, si  $f$  es un flujo de valor entero en  $G_0$ , entonces hay una coincidencia  $M$  en  $G$  con cardinalidad  $\sum_{(u, v) \in M} f(u, v)$ .

Prueba Primero mostramos que una coincidencia  $M$  en  $G$  corresponde a un flujo  $f$  de valor entero en  $G_0$ . Defina  $f$  de la siguiente manera. Si  $f(u) = 1$  para  $u \in M$ , luego  $f(u) = 0$  para  $u \in V \setminus M$ . Para todos los demás bordes  $(u, v) \in E_0$ , definimos  $f(u, v) = 0$ . Es sencillo verificar que  $f$  satisface la restricción de capacidad y la conservación del caudal.

Intuitivamente, cada arista  $.u; / 2 M$  corresponde a una unidad de flujo en  $G_0$  que recorre el camino  $s \rightarrow u \rightarrow t$ . Además, los caminos inducidos por las aristas en  $M$  son disjuntos en los vértices, excepto  $s$  y  $t$ . El flujo neto a través del corte  $.L [ fsg ; R [ ftg ]$  es igual a  $jM_j$ ; así, por el Lema 26.4, el valor del flujo es  $jf_j D jM_j$ .

Para probar lo contrario, sea  $f$  un flujo de valor entero en  $G_0$ , y deja

$\text{md } fu; / W u 2 L; 2R; y f .u; / > 0g :$

Cada vértice  $u \in 2L$  tiene sólo una arista de entrada, a saber,  $.s; u/$ , y su capacidad es 1. Por lo tanto, cada  $u \in 2L$  tiene como máximo una unidad de flujo que ingresa, y si una unidad de flujo entra, por conservación del flujo, una unidad de flujo debe salir. Además, dado que  $f$  tiene un valor entero, para cada  $u \in 2L$ , la unidad de flujo puede entrar como máximo en un borde y puede salir como máximo en un borde. Así, una unidad de flujo entra en  $u$  si y sólo si hay exactamente un vértice  $2R$  tal que  $f .u; / D 1$ , y como máximo un borde que sale de cada  $u \in 2L$  lleva flujo positivo. Un argumento simétrico se aplica a cada  $2R$ . Por lo tanto, el conjunto  $M$  es una coincidencia.

Para ver que  $jM_j D jf_j$ , observe que para cada vértice emparejado  $u \in 2L$ , tenemos  $f .s; u/ D 1$ , y para cada arista  $.u; / 2EM$ , tenemos  $f .u; / D 0$ . En consecuencia,  $f .L [ fsg ; R [ ftg /$ , el flujo neto a través del corte  $.L [ fsg ; R [ ftg /$ , es igual a  $jM_j$ . Aplicando el Lema 26.4, tenemos que  $jf_j D f .L [ fsg ; R [ ftg / D jM_j$ . ■

Con base en el Lema 26.9, nos gustaría concluir que una coincidencia máxima en un gráfico bipartito  $G$  corresponde a un flujo máximo en su correspondiente red de flujo  $G_0$  y, por lo tanto, podemos calcular una coincidencia máxima en  $G$  ejecutando un algoritmo de flujo máximo en  $G_0$ . El único inconveniente en este razonamiento es que el algoritmo de flujo máximo podría devolver un flujo en  $G_0$  para el cual algunos  $f .u; /$  no es un número entero, aunque el valor de flujo  $jf_j$  debe ser un número entero. El siguiente teorema muestra que si usamos el método de Ford-Fulkerson, esta dificultad no puede surgir.

#### Teorema 26.10 (Teorema de integralidad)

Si la función de capacidad  $c$  toma solo valores enteros, entonces el flujo máximo  $f$  producido por el método de Ford-Fulkerson tiene la propiedad de que  $jf_j$  es un número entero. Además, para todos los vértices  $u$  y el valor de  $f .u; /$  es un número entero.

Prueba La prueba es por inducción sobre el número de iteraciones. Lo dejamos como Ejercicio 26.3-2. ■

Ahora podemos probar el siguiente corolario del Lema 26.9.

**Corolario 26.11**

La cardinalidad de un máximo coincidente  $M$  en un grafo bipartito  $G$  es igual al valor de un flujo máximo  $f$  en su correspondiente red de flujo  $G_0$ .

Prueba Usamos la nomenclatura del Lema 26.9. Suponga que  $M$  es una coincidencia máxima en  $G$  y que el flujo  $f$  correspondiente en  $G_0$  no es máximo.

Entonces hay un caudal máximo  $f_j > jf$ . ~~Con Gadas que~~  $f_j$  tiene valores enteros, por el teorema 26.10, podemos suponer que  $f$  tiene valores enteros. Por lo tanto,  $f_j M_0 j D^0$  es  $jf$  coincidencia. De manera  $D^0$  corresponde a un  $M_0$  coincidente en  $G$  con cardinalidad  $j > jf$  similar,  $D^0 j M_j$ , contradiciendo nuestra suposición de que  $M$  es un máximo podemos mostrar que si  $f$  es un flujo máximo en  $G_0$ , la coincidencia correspondiente es  $D^0$ , es una coincidencia máxima en  $G$ . ■

Por lo tanto, dado un gráfico no dirigido bipartito  $G$ , podemos encontrar una coincidencia máxima creando la red de flujo  $G_0$ , ejecutando el método de Ford-Fulkerson y obteniendo directamente una coincidencia máxima  $M$  del flujo máximo de valor entero  $f$  encontrado.

Dado que cualquier coincidencia en un gráfico bipartito tiene cardinalidad como máximo  $\min(L; R/D)$ , el valor del caudal máximo en  $G_0$  es  $O(V)$ . Por lo tanto, podemos encontrar una coincidencia máxima en un gráfico bipartito en el tiempo  $O(VE^2 / D)$ , ya que  $jE_0 j D$ , es.

**Ejercicios****26.3-1**

Ejecute el algoritmo de Ford-Fulkerson en la red de flujo de la figura 26.8(c) y muestre la red residual después de cada aumento de flujo. Numere los vértices en  $L$  de arriba a abajo del 1 al 5 y en  $R$  de arriba a abajo del 6 al 9. Para cada iteración, elija la ruta de aumento que sea lexicográficamente más pequeña.

**26.3-2**

Demuestre el teorema 26.10.

**26.3-3**

Sea  $G = (V, E)$  un grafo bipartito con partición de vértices  $VL \cup VR$ , y sea  $G_0$  su red de flujo correspondiente. Proporcione un buen límite superior para la longitud de cualquier trayectoria de aumento encontrada en  $G_0$  durante la ejecución de FORD-FULKERSON.

**26.3-4 ?**

Un emparejamiento perfecto es un emparejamiento en el que se emparejan todos los vértices. Sea  $G = (V, E)$  sea un grafo bipartito no dirigido con partición de vértices  $VL \cup VR$ , donde  $|VL| = |VR|$ . Para cualquier  $X \subseteq V$ , defina la vecindad de  $X$  como

$N(X) = \{v \in V \mid \exists u \in X \text{ tal que } (u, v) \in E\}$

es decir, el conjunto de vértices adyacentes a algún miembro de X. Demostrar el teorema de Hall: existe una coincidencia perfecta en G si y sólo si  $\sum_{v \in A} d(v) \geq |A|$  para todo subconjunto A de L.

### 26.3-5 ?

Decimos que un grafo bipartito  $G = (V; E)$ , donde  $V = V_L \cup V_R$ , es d-regular si todo vértice de  $V_L$  tiene grado exactamente d. Todo grafo bipartito d-regular tiene  $|V_L| \leq |V_R|$ .

Demuestre que todo grafo bipartito d-regular tiene una coincidencia de cardinalidad  $|V_L|$  argumentando que un corte mínimo de la red de flujo correspondiente tiene capacidad  $|V_R|$ .

## ? 26.4 Algoritmos de push-relabel

En esta sección, presentamos el enfoque de "empujar-reetiquetar" para calcular los flujos máximos. Hasta la fecha, muchos de los algoritmos de flujo máximo asintóticamente más rápidos son algoritmos push-relabel, y las implementaciones reales más rápidas de algoritmos de flujo máximo se basan en el método push-relabel. Los métodos push-relabel también resuelven eficientemente otros problemas de flujo, como el problema de flujo de costo mínimo. Esta sección presenta el algoritmo de flujo máximo "genérico" de Goldberg, que tiene una implementación simple que se ejecuta en el tiempo  $O(|V|^2|E|)$ , mejorando así el límite  $O(|V|^3|E|)$  del algoritmo de Edmonds-Karp. La sección 26.5 refina el algoritmo genérico para obtener otro algoritmo push-relabel que se ejecuta en el tiempo  $O(|V|^3)$ .

Los algoritmos Push-relabel funcionan de una manera más localizada que el método Ford Fulkerson. En lugar de examinar toda la red residual para encontrar una ruta de aumento, los algoritmos de push-relabel funcionan en un vértice a la vez, observando solo los vecinos del vértice en la red residual. Además, a diferencia del método Ford Fulkerson, los algoritmos push-relabel no mantienen la propiedad de conservación del flujo a lo largo de su ejecución. Sin embargo, mantienen un preflujo, que es una función  $f: V \rightarrow \mathbb{R}$  que satisface la restricción de capacidad y la siguiente relajación de la conservación del flujo:

$$\begin{array}{c} Xf_u; u \in X \\ 2V \end{array} \quad \begin{array}{c} Xf_u; u \in X \\ 2V \end{array}$$

para todos los vértices  $u \in V$  fsg. Es decir, el flujo hacia un vértice puede exceder el flujo hacia afuera. llamamos a la cantidad

$$\begin{array}{c} eu/DX_f_u; u \in X \\ 2V \end{array} \quad \begin{array}{c} Xf_u; u \in X \\ 2V \end{array} \tag{26.14}$$

el exceso de flujo en el vértice u. El exceso en un vértice es la cantidad por la cual el flujo de entrada excede al flujo de salida. Decimos que un vértice  $u \in V$  es desbordado si  $eu > 0$ .

Comenzaremos esta sección describiendo la intuición detrás del método push-relabel. Luego investigaremos las dos operaciones empleadas por el método: "empujar" el preflujo y "re etiquetar" un vértice. Finalmente, presentaremos un algoritmo genérico push-relabel y analizaremos su corrección y tiempo de ejecución.

### Intuición

Puede comprender la intuición detrás del método push-relabel en términos de flujos de fluidos: consideramos una red de flujo  $G = (V, E)$  ser un sistema de tuberías interconectadas de capacidades dadas. Aplicando esta analogía al método de Ford-Fulkerson, podríamos decir que cada camino de aumento en la red da lugar a una corriente adicional de fluido, sin puntos de bifurcación, que fluye desde la fuente hasta el sumidero. El método de Ford-Fulkerson agrega iterativamente más corrientes de flujo hasta que no se pueden agregar más.

El algoritmo genérico push-relabel tiene una intuición bastante diferente. Como antes, los bordes dirigidos corresponden a tuberías. Los vértices, que son uniones de tuberías, tienen dos propiedades interesantes. Primero, para acomodar el exceso de flujo, cada vértice tiene una tubería de flujo de salida que conduce a un depósito arbitrariamente grande que puede acumular fluido. En segundo lugar, cada vértice, su depósito y todas sus conexiones de tuberías se asientan sobre una plataforma cuya altura aumenta a medida que avanza el algoritmo.

Las alturas de los vértices determinan cómo se empuja el flujo: solo empujamos el flujo cuesta abajo, es decir, desde un vértice más alto a un vértice más bajo. El flujo de un vértice inferior a un vértice superior puede ser positivo, pero las operaciones que empujan el flujo solo lo empujan cuesta abajo. Fijamos la altura de la fuente en  $j_V$  y la altura del sumidero en 0. Todas las demás alturas de los vértices comienzan en 0 y aumentan con el tiempo. El algoritmo primero envía la mayor cantidad de flujo posible cuesta abajo desde la fuente hacia el sumidero. La cantidad que envía es exactamente suficiente para llenar cada tubería saliente desde la fuente hasta su capacidad; es decir, envía la capacidad del corte  $s; Vfsg$ . Cuando el flujo entra por primera vez en un vértice intermedio, se acumula en el depósito del vértice. A partir de ahí, eventualmente lo empujamos cuesta abajo.

Eventualmente podemos encontrar que las únicas tuberías que salen de un vértice  $u$  y que aún no están saturadas con flujo se conectan a vértices que están en el mismo nivel que  $u$  o están cuesta arriba desde  $u$ . En este caso, para eliminar el exceso de flujo de un vértice  $u$  que se desborda, debemos aumentar su altura, una operación llamada "re etiquetado" del vértice  $u$ . Aumentamos su altura a una unidad más que la altura del más bajo de sus vecinos al que tiene una tubería no saturada. Después de volver a etiquetar un vértice, por lo tanto, tiene al menos una tubería saliente a través de la cual podemos empujar más flujo.

Eventualmente, todo el flujo que posiblemente puede llegar al fregadero ha llegado allí. No puede llegar más, porque las tuberías obedecen a las limitaciones de capacidad; la cantidad de flujo a través de cualquier corte todavía está limitada por la capacidad del corte. Para hacer que el preflujo sea un flujo "legal", el algoritmo luego envía el exceso recolectado en los depósitos de vértices desbordados de regreso a la fuente al continuar etiquetando los vértices arriba.

la altura fija  $jV$  de la fuente. Como veremos, una vez que hemos vaciado todos los embalses, el precaudal no es solo un caudal "legal", también es un caudal máximo.

### Las operaciones básicas

De la discusión anterior, vemos que un algoritmo push-relabel realiza dos operaciones básicas: empujar el exceso de flujo de un vértice a uno de sus vecinos y volver a etiquetar un vértice. Las situaciones en las que se aplican estas operaciones dependen de las alturas de los vértices, que ahora definiremos con precisión.

Sea  $G(V, E)$ ; Sea  $s$  una fuente y  $t$  un sumidero, y sea  $f$  un preflujo en  $G$ . Una función  $h: V \rightarrow \mathbb{N}$  es una función de altura<sup>3</sup> si  $h(s) = 0$ ,  $h(t) = D$ , y

$h(u) < h(v)$  para cada arista residual  $(u, v) \in E$ .

para cada arista residual  $(u, v) \in E$ . Inmediatamente obtenemos el siguiente lema.

### Lema 26.12

Sea  $G(V, E)$ ; Sea  $E$  una red de flujo, sea  $f$  un preflujo en  $G$ , y sea  $h$  una función de altura en  $V$ . Para cualesquiera dos vértices  $u, v \in V$  si  $h(u) > h(v)$ , entonces  $(u, v) \in E$  no es un borde en la red residual. ■

### La operación push

La operación básica  $PUSH(u)$  se aplica si  $u$  es un vértice desbordante,  $c_f(u) > 0$ , y  $h(u) < D$ . El pseudocódigo a continuación actualiza el preflujo  $f$  y los flujos en exceso para  $u$  y asume que podemos calcular la capacidad residual  $c_f(u)$  en tiempo constante dadas  $c$  y  $f$ . Mantenemos el exceso de flujo almacenado en un vértice  $u$  como el atributo  $u.e$  y la altura de  $u$  como el atributo  $u.h$ . La expresión  $f(u)$  es una variable temporal que almacena la cantidad de flujo que podemos empujar desde  $u$  hasta .

<sup>3</sup>En la bibliografía, una función de altura se suele denominar "función de distancia" y la altura de un vértice se denomina "etiqueta de distancia". Usamos el término "altura" porque sugiere más la intuición detrás del algoritmo. Mantenemos el uso del término "re etiquetar" para referirnos a la operación que aumenta la altura de un vértice. La altura de un vértice está relacionada con su distancia desde el sumidero  $t$ , como se encontraría en una búsqueda en anchura de la transpuesta GT.

```

PUSH.u; /
1 // Se aplica cuando: u se desborda, cf .u; / > 0, y u:h D :h C 1.
2 // Acción: Empuje f .u; / D min.u:e; cf .u; // unidades de flujo de u a . 3 f .u; / D min.u:e;
cf .u; // 4 si .u; / 2 E 5 .u; //f D .u; //f C
f .u; / 6 más .; u:/f
D .; u:/f f .u; / 7 u:e D u:e f .u; / 8 :e D :e C
f .u; /

```

El código para PUSH funciona de la siguiente manera. Como el vértice  $u$  tiene un exceso positivo  $u:e$  y la capacidad residual de  $.u;$  es positivo, podemos aumentar el flujo de  $u$  a por  $f .u; / D \min.u:e; cf .u;$  // sin que  $u:e$  se vuelva negativo o la capacidad  $cu;$  / ser superado. La línea 3 calcula el valor  $f .u; /$ , y las líneas 4 a 6 actualizan  $f$ . La línea 5 aumenta el flujo en el borde  $.u; /$ , porque estamos empujando el flujo sobre un borde residual que también es un borde original. La línea 6 disminuye el flujo en el borde  $.; u/$ , porque el borde residual es en realidad el reverso de un borde en la red original. Finalmente, las líneas 7 y 8 actualizan los flujos en exceso en los vértices  $u$  y Por lo tanto, si  $f$  es un preflujo antes de llamar a PUSH , sigue siendo un preflujo después.

Observe que nada en el código para PUSH depende de las alturas de  $u$  y, sin embargo, prohibimos que se invoque a menos que  $u:h D :h C 1$ . Por lo tanto, empujamos el exceso de flujo cuesta abajo solo por un diferencial de altura de 1. Por el Lema 26.12 , no existen aristas residuales entre dos vértices cuyas alturas difieren en más de 1 y, por lo tanto, mientras el atributo  $h$  sea de hecho una función de altura, no ganaríamos nada al permitir que el flujo sea empujado cuesta abajo por una diferencia de altura de más de 1 .

Llamamos a la operación  $PUSH.u; /$  un empunjón de  $u$  a . Si se aplica una operación de empuje a algún borde  $.u; /$  dejando un vértice  $u$ , también decimos que la operación de empujar se aplica a  $u$ . Es un empunjón de saturación si  $edge .u; /$  en la red residual se satura ( $cf .u; / D 0$  después); de lo contrario, es un empunjón no saturado. Si un borde se satura, desaparece de la red residual. Un lema simple caracteriza un resultado de un empunjón no saturado.

#### Lema 26.13

Después de un empunjón no saturado de  $u$  a , el vértice  $u$  ya no se desborda.

Prueba Dado que el empuje no fue saturado, la cantidad de flujo  $f .u; /$  realmente empujado debe ser igual a  $u:e$  antes del empuje. Dado que  $u:e$  se reduce en esta cantidad, se convierte en 0 después del empuje. ■

La operación de

re etiquetado La operación básica RELABEL.u/ se aplica si u se desborda y si  $u:h > h$  para todos los bordes  $.u; / 2 E$  En otras palabras, podemos volver a etiquetar un vértice desbordante u si por cada vértice para el que hay capacidad residual de  $u$  a fluir no se puede empujar de  $u$  a porque no está cuesta abajo desde  $u$ . (Recuerde que, por definición, ni la fuente  $s$  ni el sumidero  $t$  pueden desbordarse, por lo que  $s$  y  $t$  no son elegibles para volver a etiquetarse).

RELABEL.u/

1 // Se aplica cuando:  $u$  se desborda y para todo  $2 V$  tal que  $.u; / 2 E$ , tenemos  $u:h > h$ .

2 // Acción: Aumentar la altura de  $u$ . 3  $u:h$

$D 1 C \min f:h W .u; / 2 E$

Cuando llamamos a la operación RELABEL.u/, decimos que el vértice  $u$  está re etiquetado. Tenga en cuenta que cuando se vuelve a etiquetar  $u$ ,  $E$  debe contener al menos un borde que deja  $u$ , de modo que la minimización en el código sea sobre un conjunto no vacío. Esta propiedad se deriva de la suposición de que  $u$  se está desbordando, lo que a su vez nos dice que

$$\begin{array}{c} u:e \\ 2V \end{array} \leq \begin{array}{c} DX f .; u/ \\ 2V \end{array} \leq \begin{array}{c} X f .u; / \\ 2V \end{array} > 0$$

Dado que todos los flujos son no negativos, debemos tener al menos un vértice tal que  $.; u:f > 0$ . Pero entonces,  $cf .u; / > 0$ , lo que implica que  $.u; / 2 E$ . La operación RELABEL.u/ da a  $u$  la mayor altura permitida por las restricciones de las funciones de altura.

El algoritmo genérico

El algoritmo genérico push-relabel utiliza la siguiente subrutina para crear un preflujo inicial en la red de flujo.

```
INICIALIZAR-PREFLUJO.G; s/
 1 para cada vértice 2 G:V 2 :h
  D 0 3 :e D 0 4
    para cada
      arista  $.u; / 2 G:E 5 .u; /:f$  D 0 6 s:h
      D jG:Vj 7 para cada
        vértice 2 s:Adj .s; //
        f D cs; / 8 :e D cs; / s:e D s:e
          cs; /
 9
10
```

INITIALIZE-PREFLOW crea un preflujo inicial  $f$  definido por

.u; /f D ( cu; / si u D s de lo contrario : (26.15)

Es decir, llenamos al máximo cada borde dejando la fuente  $s$ , y todos los demás bordes no llevan flujo. Para cada vértice adyacente a la fuente, inicialmente tenemos  $\text{eD}(cs) = \emptyset$ , e inicializamos  $s := -\infty$ . El algoritmo genérico también comienza con una función de altura inicial  $h$ , dada por

u h D ( IV i si u D s ) (26.16)

La ecuación (26.16) define una función de altura porque las únicas aristas  $u; /$  para los cuales  $u:h > :h C 1$  son aquellos para los cuales  $u \in D_s$ , y esos bordes están saturados, lo que significa que no están en la red residual.

Inicialización, seguida de una secuencia de operaciones de inserción y reetiquetado, ejecutada sin ningún orden en particular, produce el algoritmo **GENERIC-PUSH-RELABEL**:

GENERIC-PUSH-REF ABEI\_G/ 1

INITIALIZE-PREVIOUS-G: s / 2 mientras

exista una operación push o reetiquetado aplicable.

seleccione una operación de inserción o reetiquetado aplicable y llévela a cabo.

El siguiente lema nos dice que mientras exista un vértice desbordante, se aplica al menos una de las dos operaciones básicas.

Lema 26.14 (Un vértice desbordado se puede empujar o volver a etiquetar)

Sea  $G = (V, E)$  una red de flujo con fuente  $s$  y sumidero  $t$ , sea  $f$  un preflujo y sea  $h$  cualquier

función de altura para  $f$ .

Prueba Para cualquier borde residual  $u; l$ , tenemos  $hu/ h \leq C_1$  porque  $h$  es una función de altura. Si una operación de empuje no se aplica a un vértice desbordante  $u$ , entonces para todos los bordes residuales  $u; l$ , debemos tener  $hu/ h \leq C_1$ , lo que implica  $hu/ h \leq C_1$ . Por lo tanto, se aplica una operación de rectificadura a  $u$ .

#### Corrección del método push relativo

Para mostrar que el algoritmo genérico push-relabel resuelve el problema del flujo máximo, primero probaremos que si termina, el prefijo  $f$  es un flujo máximo.

Más adelante probaremos que termina. Comenzamos con algunas observaciones sobre la función altura  $b$ .

Lema 26.15 (La altura de los vértices nunca disminuye)

Durante la ejecución del procedimiento GENERIC-PUSH-RELABEL sobre una red de flujo  $G = \langle V, E \rangle$ , para cada vértice  $u \in V$  la altura  $u:h$  nunca decrece. Además, cada vez que se aplica una operación de reetiquetado a un vértice  $u$ , su altura  $u:h$  aumenta en al menos 1.

Prueba Debido a que las alturas de los vértices cambian solo durante las operaciones de reetiquetado, es suficiente probar la segunda afirmación del lema. Si el vértice  $u$  está a punto de ser relabelado, entonces para todos los vértices tales que  $u; / 2 \text{ Ef}$ , tenemos  $u:h \geq h$ . Así,  $u:h < 1 \leq \min f:h \leq W \leq u; / 2 \text{ Ef} g$ , por lo que la operación debe aumentar  $u:h$ . ■

Lema 26.16 Sea

$G = \langle V, E \rangle$  sea una red de flujo con fuente  $s$  y sumidero  $t$ . Luego, la ejecución de GENERIC-PUSH-RELABEL en  $G$  mantiene el atributo  $h$  como una función de altura.

Demostración La demostración es por inducción sobre el número de operaciones básicas realizadas. Inicialmente,  $h$  es una función de altura, como ya hemos observado.

Decimos que si  $h$  es una función de altura, entonces una operación RELABEL. $u;$  deja una función de altura. Si nos fijamos en una arista residual  $u; / 2$  Si eso deja  $u$ , entonces la operación RELABEL. $u;$  asegura que  $u:h \geq h + 1$  después. Ahora considere una arista residual  $w; u;$  que entra  $u$ . Por el Lema 26.15,  $w:h \leq u:h \leq h + 1$  antes de la operación RELABEL. $u;$  implica  $w:h < u:h \leq h + 1$  después. Así, la operación RELABEL. $u;$  deja la función de altura.

Ahora, considere una operación PUSH. $u; /$ . Esta operación puede agregar el borde  $u; /$  y puede tener  $h$  es una función de altura. En este último caso, eliminando  $u; /$  de la red residual elimina la restricción correspondiente, y  $h$  nuevamente sigue siendo una función de altura. ■

El siguiente lema da una propiedad importante de las funciones de altura.

Lema 26.17 Sea

$G = \langle V, E \rangle$  sea  $E$  una red de flujo con fuente  $s$  y sumidero  $t$ , sea  $f$  un preflujo en  $G$ , y sea  $h$  una función de altura en  $V$ . Entonces no hay camino desde la fuente  $s$  hasta el sumidero  $t$  en la red residual  $G_f$ .

Prueba Asumir por el bien de la contradicción que  $G_f$  contiene un camino  $p$  de  $s$  a  $t$ , donde  $p$  no tiene vértices de altura menor que  $h$ . Sin pérdida de generalidad,  $p$  es un camino simple, por lo que  $k < j$  para todo  $j \in p$ . Para todo  $i \in p$ ,  $i; / : k$ , borde  $i; /$  es de altura  $h_i$ .

Como  $h$  es una función de altura,  $h_i \leq h_j$  para todo  $i; / : k$ , borde  $i; /$  es de altura  $h_i$ . La combinación de estas desigualdades sobre el camino  $p$  produce  $h_s \leq h_t$ . Pero debido a que  $h_t < h$ ,

tenemos  $hs/k < jV_j$ , lo que contradice el requisito de que  $hs/D \geq jV_j$  en una función de altura.

■

Ahora estamos listos para mostrar que si el algoritmo genérico push-relabel termina, el preflujo que calcula es un flujo máximo.

**Teorema 26.18 (Corrección del algoritmo genérico push-relabel)**

Si el algoritmo GENERIC-PUSH-RELABEL termina cuando se ejecuta en una red de flujo  $G = (V, E)$  con fuente  $s$  y sumidero  $t$ , entonces el preflujo  $f$  que calcula es un flujo máximo para  $G$ .

Prueba Usamos el siguiente bucle invariante:

Cada vez que se ejecuta la prueba de ciclo while en la línea 2 en GENERIC-PUSH-RELABEL ,  $f$  es un flujo previo.

Inicialización: INITIALIZE-PREFLOW hace un preflujo.

Mantenimiento: Las únicas operaciones dentro del bucle while de las líneas 2 y 3 son empujar y volver a etiquetar. Las operaciones de reetiquetado afectan solo a los atributos de altura y no a los valores de flujo; por lo tanto, no afectan si  $f$  es un preflujo. Como se argumentó en la página 739, si  $f$  es un preflujo antes de una operación de empuje, sigue siendo un preflujo después.

Terminación: En la terminación, cada vértice en  $V \setminus \{s, t\}$  debe tener un exceso de 0, porque por el Lema 26.14 y el invariante de que  $f$  es siempre un preflujo, no hay vértices desbordantes. Por lo tanto,  $f$  es un flujo. El Lema 26.16 muestra que  $h$  es una función de altura en la terminación y, por lo tanto, el Lema 26.17 nos dice que no hay un camino de  $s$  a  $t$  en la red residual  $G_f$ . Por el teorema de corte mínimo de flujo máximo (Teorema 26.6), por lo tanto,  $f$  es un flujo máximo.

■

### Análisis del método push-relabel

Para mostrar que el algoritmo genérico push-relabel realmente termina, limitaremos el número de operaciones que realiza. Vinculamos por separado cada uno de los tres tipos de operaciones: reetiquetado, inserciones de saturación y inserciones de no saturación. Con el conocimiento de estos límites, es un problema sencillo construir un algoritmo que se ejecute en el tiempo  $O(V^2E)$ . Sin embargo, antes de comenzar el análisis, probaremos un lema importante. Recuerde que permitimos bordes en la fuente en la red residual.

**Lema 26.19 Sea**

$G = (V, E)$  una red de flujo con fuente  $s$  y sumidero  $t$ , y sea  $f$  un preflujo en  $G$ . Entonces, para cualquier vértice desbordante  $x$ , hay un camino simple de  $x$  a  $s$  en la red residual  $G_f$ .

Demostración Para un vértice desbordante  $x$ , sea  $UD \setminus W$  que existe un camino simple desde  $x$  hasta en  $G \setminus g$ , y supongamos, en aras de la contradicción, que  $s \in 2U$ . Sea  $UDV \setminus U$ .

Tomamos la definición de exceso de la ecuación (26.14), sumamos todos los vértices en  $U$  y observamos que  $VDU \setminus U$ , para obtener

$$\begin{matrix} X & UE/ \\ u_{2u} & \end{matrix}$$

$$\begin{matrix} DXX f.; tu/X \\ u_{2u} & 2V \\ & 2V f.u; /! \end{matrix}$$

$$\begin{matrix} DX & Xf.; tu/CX \\ u_{2u} & 2U \\ & 2\bar{U} f.; tu/! X \\ & 2U f.u; /CX \\ & 2\bar{U} f.u; /!! \end{matrix}$$

$$\begin{matrix} DXX f.; u/CXX f.; u/ XX f.u; /XX \\ u_{2u} & 2U \\ u_{2u} & 2\bar{U} \\ & u_{2u} & 2U \\ & u_{2u} & 2\bar{U} f.u; / \end{matrix}$$

$$\begin{matrix} DXX f.; u/ XX f.u; / : \\ u_{2u} & 2\bar{U} \\ u_{2u} & 2U \end{matrix}$$

Sabemos que la cantidad  $P_{u_{2u}} e_u$  debe ser positivo porque  $e_x > 0$ ,  $x \in 2U$ , todos los vértices excepto  $s$  tienen un exceso no negativo y, por suposición,  $s \in 2U$ . Así, tenemos

$$\begin{matrix} XXf.; u/ XX f.u; / > 0 \\ u_{2u} & 2\bar{U} \\ u_{2u} & 2U \end{matrix} \quad (26.17)$$

Todos los flujos de borde son no negativos, por lo que para que se cumpla la ecuación (26.17), debemos tener  $PP f.; u > 0$ . Por tanto, debe existir al menos un par de vértices  $u_0 \in 2\bar{U}$  y  $2U$  con  $f.u_0 > 0$ . Pero, si  $f.u_0 > 0$ , debe haber un (el borde residual  $u_0$  ; /), lo que significa que hay un camino simple de  $x$  ruta  $x - u_0 - a$ , lo que contradice la definición de  $U$ . ■

El siguiente lema limita las alturas de los vértices y su corolario limita el número de operaciones de reetiquetado que se realizan en total.

### Lema 26.20

Sea  $GD \setminus V; E$  una red de flujo con fuente  $s$  y sumidero  $t$ . En cualquier momento durante la ejecución de GENERIC-PUSH-RELABEL en  $G$ , tenemos  $u:h \leq j \leq j+1$  para todos los vértices  $u \in 2V$ .

Prueba Las alturas de la fuente  $s$  y el sumidero  $t$  nunca cambian porque estos vértices, por definición, no se desbordan. Por lo tanto, siempre tenemos  $s:h = D \leq j \leq j+1$  y  $t:h = 0$ , los cuales no son mayores que  $2j$ .

Ahora considere cualquier vértice  $u \in 2V$   $f_s(tg)$ . Inicialmente,  $u:h = D \leq j \leq j+1$ . Deberíamos Demuestre que después de cada operación de reetiquetado, todavía tenemos  $u:h = j$ . Cuando estás

re etiquetado, se desborda, y el Lema 26.19 nos dice que hay un camino simple  $p$  de  $u$  a  $s$  en  $G_f$ . Sea  $p: D_0; 1; \dots; k_i$ , donde  $D_s$ , y  $k \geq j+1$  porque  $p$  es simple. Para  $i: D_0; 1; \dots; k_i$ , tenemos  $i; iC_1; 2E_f$ , y por lo tanto, por el Lema 26.16,  $i: h_iC_1; h_iC_1$ . Al expandir estas desigualdades sobre  $k: h_iC_1 \leq h_iC_1 \cdot jV_j / 1 / D_2 \geq jV_j$  camino  $p$  se obtiene  $u: h_iD_0 \geq h_iC_1$ .

#### Corolario 26.21 (Consolidado en operaciones de re etiquetado)

Sea  $G: V; E$  una red de flujo con fuente  $s$  y sumidero  $t$ . Entonces, durante la ejecución de GENERIC-PUSH-RELABEL en  $G$ , el número de operaciones de re etiquetado es como máximo  $2 \cdot jV_j + 1$  por vértice y como máximo  $2 \cdot jV_j + 1 / jV_j + 2 / < 2 \cdot jV_j^2$  en general.

Prueba Sólo los vértices  $jV_j + 2$  en  $V_{fs;tg}$  pueden ser re etiquetados. Sea  $u: 2V_{fs;tg}$ . La operación  $RELABEL(u)$  aumenta  $u: h$ . El valor de  $u: h$  es inicialmente 0 y por el Lema 26.20 crece hasta  $2 \cdot jV_j + 1$  como máximo. Por lo tanto, cada vértice  $u: 2V_{fs;tg}$  se vuelve a etiquetar como máximo  $2 \cdot jV_j + 1$  veces, y el número total de re etiquetados operaciones realizadas es como máximo  $2 \cdot jV_j + 1 / jV_j + 2 / < 2 \cdot jV_j^2$ .

El lema 26.20 también nos ayuda a acotar el número de empujones de saturación.

#### Lema 26.22 (Limitado en empujones de saturación)

Durante la ejecución de GENERIC-PUSH-RELABEL sobre cualquier red de flujo  $G: V; E$ , el número de impulsos de saturación es inferior a  $2 \cdot jV_j + jE_j$ .

Prueba Para cualquier par de vértices  $u: 2V$  contaremos los empujones de saturación de  $u$  hacia y de  $a$  juntos, llamándolos empujones de saturación entre  $u$ . Si hay tales empujones, al menos uno de  $u: y$ ;  $u: y$  es en realidad y una arista en  $E$ . Ahora, suponga que ha ocurrido un empuje de saturación desde  $u$  hasta.

En ese momento,  $:h D u: h 1$ . Para que ocurra otro empuje de  $u$  a más tarde, el algoritmo primero debe empujar el flujo de  $u$  a  $u$ , lo que no puede suceder hasta que  $:h D u: h C 1$ . Dado que  $u: h$  nunca disminuye, para que  $:h D u: h C 1$ , el valor de  $:h$  debe aumentar en al menos 2. Del mismo modo,  $u: h$  debe aumentar en al menos 2 entre empujones de saturación desde  $u$  a  $u$ . Las alturas comienzan en 0 y, por el Lema 26.20, nunca exceden  $2 \cdot jV_j + 1$ , lo que implica que el número de veces que cualquier vértice puede aumentar su altura en 2 es menor que  $jV_j$ . Dado que al menos uno de  $u: h$  y  $:h$  debe aumentar en 2 entre cualquiera de los dos empujones de saturación entre  $u$  y  $y$  hay menos de  $2 \cdot jV_j$  empujones de saturación entre  $u$  y  $y$ . Multiplicar por el número de aristas da un límite de menos de  $2 \cdot jV_j + jE_j$  en el número total de impulsos de saturación.

El siguiente lema limita el número de pulsaciones no saturadas en el algoritmo de re etiquetado de pulsaciones genérico.

Lema 26.23 (Límite en empujones no saturados)

Durante la ejecución de GENERIC-PUSH-RELABEL en cualquier red de flujo GD

$.V; E/$ , el número de impulsos no saturados es inferior a  $4 jV j^2 .jV jC jEj/$ .

Prueba Defina una función de potencial  $\hat{D}P :h$ . Inicialmente,  $\hat{D} 0$ , y el valor de  $\hat{D}$  puede cambiar después de cada reetiquetado, impulso de saturación y impulso de no saturación. Acotaremos la cantidad que los empujones de saturación y los reetiquetamientos pueden contribuir al aumento de  $\hat{D}$ . Luego mostraremos que cada impulso no saturado debe disminuir  $\hat{D}$  en al menos 1, y usaremos estos límites para derivar un límite superior en el número de impulsos no saturados.

Examinemos las dos formas en que  $\hat{D}$  podría aumentar. Primero, reetiquetar un vértice u aumenta  $\hat{D}$  en menos de  $2 jV j$ , ya que el conjunto sobre el que se toma la suma es el mismo y el reetiquetado no puede aumentar la altura de u más que su altura máxima posible, que, por el Lema 26.20, está en la mayoría  $2 jV j 1$ . En segundo lugar, un empujón de saturación de un vértice u a un vértice aumenta  $\hat{D}$  en menos de  $2 jV j$ , ya que las alturas no cambian y solo el vértice cuya altura es como máximo  $2 jV j 1$ , posiblemente se desborde.

Ahora mostramos que un empuje no saturado de u a disminuye  $\hat{D}$  en al menos 1.

¿Por qué? Antes del impulso de no saturación, u se estaba desbordando, y puede que se haya desbordado o no. Por el Lema 26.13, u ya no se desborda después del empuje. Además, a menos que sea la fuente, puede o no estar rebosante después del empujón. Por lo tanto, la función de potencial  $\hat{D}$  ha disminuido exactamente en  $u:h$  y ha aumentado en  $0$  o en  $:h$ . Dado que  $u:h :h D 1$ , el efecto neto es que la función potencial ha disminuido en al menos 1.

Por lo tanto, durante el curso del algoritmo, la cantidad total de aumento en  $\hat{D}$  se debe a reetiquetados y empujes saturados, y el Corolario 26.21 y el Lema 26.22 restringen el aumento a menos de  $.2 jV j/.2 jV j / C .2 jV j/.2 jV j jEj/ D 4 jV j .jV j C jEj/$ . Como  $\hat{D} 0$ , la cantidad total de disminución  $\hat{D}$  y, por lo tanto, el número total de impulsos no saturados, es menor que  $4 jV j .jV j C jEj/$ . ■

Habiendo acotado el número de reetiquetados, impulsos de saturación y impulsos de clasificación no saturados, hemos preparado el escenario para el siguiente análisis del procedimiento GENERIC PUSH-RELABEL y, por lo tanto, de cualquier algoritmo basado en el método push-relabel.

#### Teorema 26.24

Durante la ejecución de GENERIC-PUSH-RELABEL sobre cualquier red de flujo GD  $.V; E/$ , el número de operaciones básicas es  $OV 2E/$ .

Prueba Inmediata del Corolario 26.21 y los Lemas 26.22 y 26.23. ■

Por lo tanto, el algoritmo termina después de las operaciones OV 2E/. Todo lo que queda es dar un método eficiente para implementar cada operación y para elegir una operación apropiada para ejecutar.

#### Corolario 26.25

Existe una implementación del algoritmo genérico push-relabel que se ejecuta en OV 2E/ tiempo en cualquier red de flujo GD .V; MI/.

El ejercicio de prueba 26.4-2 le pide que muestre cómo implementar el algoritmo genérico con una sobrecarga de OV / por operación de reetiquetado y O.1/ por inserción. También le pide que diseñe una estructura de datos que le permita elegir una operación aplicable en el tiempo O.1/. Luego sigue el corolario. ■

#### Ejercicios

##### 26.4-1

Demostrar que, después del procedimiento INITIALIZE-PRFLOW.G; s/ termina, tenemos  $j_f \geq j$ ,  
s: e donde f es un flujo máximo para G.

##### 26.4-2

Muestre cómo implementar el algoritmo genérico push-relabel usando OV / tiempo por operación de reetiquetado, O.1/ time per push y O.1/ time para seleccionar una operación aplicable, por un tiempo total de OV 2E / .

##### 26.4-3

Demuestre que el algoritmo genérico push-relabel gasta un total de solo O.VE/ tiempo en realizar todas las operaciones OV 2/ de reetiquetado.

##### 26.4-4

Suponga que hemos encontrado un flujo máximo en una red de flujo GD .V; E/ utilizando un algoritmo push-relabel. Proporcione un algoritmo rápido para encontrar un corte mínimo en G.

##### 26.4-5

Proporcione un algoritmo push-relabel eficiente para encontrar una coincidencia máxima en un gráfico bipartito. Analiza tu algoritmo.

##### 26.4-6

Suponga que todas las capacidades de borde en una red de flujo GD .V; E/ están en el conjunto  $f_1; 2; \dots; k$ . Analice el tiempo de ejecución del algoritmo genérico push-relabel en términos de  $|V| j, |E| j$  y  $k$ . (Pista: ¿cuántas veces puede cada borde soportar un impulso no saturado antes de saturarse?)

## 26.4-7

Muestre que podemos cambiar la línea 6 de INITIALIZE-PRFLOW a

$6 \ s:h \ D \ jG:Vj \ 2$

sin afectar la corrección o el rendimiento asintótico del algoritmo de reetiquetado de inserción genérico.

## 26.4-8

Sea  $\text{if } .u; /$  ser la distancia (número de aristas) desde  $u$  hasta en la red residual  $G_f$ . Demuestre que el procedimiento GENERIC-PUSH-RELABEL mantiene las propiedades de que  $u:h < jV j$  implica  $u:h \text{ if } .u; t /$  y que  $u:h jV j$  implica  $u:h jV j \text{ if } .u; s/$ .

## 26.4-9 ?

Como en el ejercicio anterior, sea  $\text{if } .u; /$  ser la distancia de  $ua$  en la red residual  $G_f$ . Muestre cómo modificar el algoritmo genérico push-relabel para mantener la propiedad de que  $u:h < jV j$  implica  $u:h D \text{ if } .u; t /$  y que  $u:h jV j$  implica  $u:h jV j D \text{ if } .u; s/$ . El tiempo total que su implementación dedica a mantener esta propiedad debe ser  $O(VE)$ .

## 26.4-10

Muestre que el número de impulsos no saturados ejecutados por el procedimiento GENERIC-PUSH RELABEL en una red de flujo  $GD$  . $V; E/$  es como máximo  $4 jV j jV j^2$  Ej para 4.

## ? 26.5 El algoritmo de reetiquetado al frente

El método push-relabel nos permite aplicar las operaciones básicas en cualquier orden. Sin embargo, al elegir el orden con cuidado y administrar la estructura de datos de la red de manera eficiente, podemos resolver el problema de flujo máximo más rápido que el límite  $OV 2E/$  dado por el Corolario 26.25. Ahora examinaremos el algoritmo de reetiquetado al frente, un algoritmo de reetiquetado empujado cuyo tiempo de ejecución es  $OV 3/$ , que es asintóticamente al menos tan bueno como  $OV 2E/$ , e incluso mejor para redes densas.

El algoritmo de reetiquetado al frente mantiene una lista de los vértices en la red. Comenzando por el frente, el algoritmo escanea la lista, seleccionando repetidamente un vértice desbordante  $u$  y luego "descargando", es decir, realizando operaciones de inserción y reetiquetado hasta que  $u$  ya no tiene un exceso positivo. Cada vez que volvemos a etiquetar un vértice, lo movemos al frente de la lista (de ahí el nombre "reetiquetar al frente") y el algoritmo comienza su escaneo nuevamente.

La corrección y el análisis del algoritmo de reetiquetado al frente dependen de la noción de bordes "admisibles": esos bordes en la red residual a través de los cuales se puede empujar el flujo. Después de probar algunas propiedades sobre la red de bordes admisibles, investigaremos la operación de descarga y luego presentaremos y analizaremos el propio algoritmo de reetiquetado al frente.

### Bordes y redes admisibles

Si  $G = \langle V, E \rangle$  es una red de flujo con fuente  $s$  y sumidero  $t$ ,  $f$  es un preflujo en  $G$ , y  $h$  es una función de altura, entonces decimos que  $e = (u, v) \in E$  es una arista admisible si  $f(e) > 0$  y  $h(v) - h(u) \leq 1$ . En caso contrario,  $e = (u, v)$  es inadmisible. La red admisible es  $G_f = \langle V, E_f \rangle$ , donde  $E_f$  es el conjunto de aristas admisibles.

La red admisible consta de esos bordes a través de los cuales podemos empujar el flujo. El siguiente lema muestra que esta red es un grafo acíclico dirigido (dag).

### Lema 26.26 (La red admisible es acíclica)

Si  $G = \langle V, E \rangle$  es una red de flujo,  $f$  es un preflujo en  $G$  y  $h$  es una función de altura en  $G$ , entonces la red admisible  $G_f = \langle V, E_f \rangle$  es acíclico.

Prueba La prueba es por contradicción. Suponga que  $G_f$  contiene un ciclo  $C = v_0, v_1, \dots, v_k, v_0$ , donde  $v_i \neq v_j$  para  $i \neq j$ . Sumar alrededor del ciclo da

$$\begin{array}{c} k \quad k \\ X h(v_0) / DX h(v_1) / C 1 / \\ iD1 \quad iD1 \\ \quad k \\ DX h(v_k) / C k \\ \quad iD1 \end{array}$$

Debido a que cada vértice en el ciclo  $C$  aparece una vez en cada una de las sumas, derivamos la contradicción de que  $0 \neq h(v_0) - h(v_0)$ . ■

Los siguientes dos lemas muestran cómo las operaciones de inserción y reetiquetado cambian la red admisible.

### Lema 26.27

Sea  $G = \langle V, E \rangle$  una red de flujo, sea  $f$  un preflujo en  $G$ , y supongamos que el atributo  $h$  es una función de altura. Si un vértice  $u$  se desborda y  $e = (u, v) \in E$  es una arista admisible, entonces  $PUSH(u, v)$  se aplica. La operación no crea ningún nuevo borde admisible, pero puede causar  $e = (v, u)$  convertirse en inadmisible.

Prueba Por la definición de un borde admisible, podemos empujar el flujo de  $u$  a  $.$

Como  $u$  se desborda, la operación PUSH. $u$ ; / se aplica. El único borde residual nuevo que se puede crear empujando el flujo de  $u$  a es  $; u/$ . Dado que  $:h D u:h 1$ , arista  $; u/$  no puede ser admisible. Si la operación es un empujón de saturación, entonces  $cf .u/ ; D 0$  después y  $.u/$ ; / se vuelve inadmisible.

■

#### Lema 26.28 Sea

GD .V; Sea  $E$  una red de flujo, sea  $f$  un preflojo en  $G$ , y supongamos que el atributo  $h$  es una función de altura. Si un vértice  $u$  se desborda y no hay aristas admisibles que salgan de  $u$ , entonces se aplica RELABEL. $u/$ . Despues de la operación de reetiquetado, hay al menos un borde admisible que sale de  $u$ , pero no hay bordes admisibles que entren en  $u$ .

Prueba Si  $u$  se desborda, entonces por el Lema 26.14, se le aplica una operación de empujar o reetiquetar. Si no hay aristas admisibles que salgan de  $u$ , entonces no se puede empujar flujo desde  $u$ , por lo que se aplica RELABEL. $u/$ . Despues de la operación de reetiquetado,  $u:h D 1 C \min f:h W .u/ ; 2$  Ef g. Así, si es un vértice el que realiza el mínimo en este conjunto, la arista  $.u/$ ; / se vuelve admisible. Por lo tanto, despues de la reetiqueta, hay al menos una arista admisible que sale de  $u$ .

Para mostrar que no entran aristas admisibles en  $u$  despues de una operación de reetiquetado, suponga que hay un vértice tal que  $; u/$  es admisible. Luego,  $:h D u:h C 1$  despues de la nueva etiqueta, y así  $:h > u:h C 1$  justo antes de la nueva etiqueta. Pero por el Lema 26.12, no existen aristas residuales entre vértices cuyas alturas difieren en más de 1. Además, volver a etiquetar un vértice no cambia la red residual. De este modo,  $; u/$  no está en la red residual, y por lo tanto no puede estar en la red admisible.

■

#### Listas de vecinos

Los bordes en el algoritmo de reetiquetado al frente se organizan en "listas de vecinos". Dada una red de flujo GD .V;  $E$ , la lista de vecinos  $u:N$  para un vértice  $u \in V$  es una lista unida de los vecinos de  $u$  en  $G$ . Así, el vértice aparece en la lista  $u:N$  si  $.u/ ; 2 E$  o  $; u/ 2 E$ . La lista de vecinos  $u:N$  contiene exactamente aquellos vértices para los que puede haber una arista residual  $.u/$ . El atributo  $u:N:head$  apunta al primer vértice en  $u:N$ , y  $:next-neighbor$  apunta al vértice siguiente en una lista de vecinos; este puntero es NIL si es el último vértice en la lista de vecinos.

El algoritmo de reetiquetado al frente recorre cada lista de vecinos en un orden arbitrario que se fija a lo largo de la ejecución del algoritmo. Para cada vértice  $u$ , el atributo  $u:actual$  apunta al vértice actualmente bajo consideración en  $u:N$ .

Inicialmente,  $u:current$  se establece en  $u:N:head$ .

### Descarga de un vértice desbordante

Un vértice  $u$  desbordado se descarga empujando todo su exceso de flujo a través de los bordes admisibles hacia los vértices vecinos, reetiquetando  $u$  según sea necesario para hacer que los bordes que dejan  $u$  se vuelvan admisibles. El pseudocódigo es el siguiente.

```

DESCARGA.u/ 1
mientras que  $u:e > 0$ 
2           D u: actual
3           if == NIL
4           RELABEL.u/
5           u:current D u:N:head elseif
6           cf .u; / > 0 y u:h == :h C 1 PUSH.u; / else u:actual
7           D :próximo-
8           vecino

```

La figura 26.9 recorre varias iteraciones del bucle while de las líneas 1 a 8, que se ejecuta siempre que el vértice  $u$  tenga un exceso positivo. Cada iteración realiza exactamente una de tres acciones, dependiendo del vértice actual en la lista de vecinos  $u:N$ .

1. Si es NIL, entonces hemos llegado al final de  $u:N$ . La línea 4 vuelve a etiquetar el vértice  $u$ , y luego la línea 5 restablece el vecino actual de  $u$  para que sea el primero en  $u:N$ .  
(El lema 26.29 a continuación establece que la operación de reetiquetado se aplica en esta situación).
2. Si no es NIL y  $.u; /$  es un borde admisible (determinado por la prueba en la línea 6), luego la línea 7 empuja algo (o posiblemente todo) del exceso de  $u$  al vértice.
3. Si no es NIL sino  $.u; /$  es inadmisible, luego la línea 8 avanza  $u: actual$  posición más adelante en la lista de vecinos  $u:N$ .

Observe que si se llama DESCARGA en un vértice  $u$  desbordado, entonces la última acción realizada por DESCARGA debe ser un empujón de  $u$ . ¿Por qué? El procedimiento termina solo cuando  $u:e$  se vuelve cero, y ni la operación de reetiquetado ni el avance del puntero  $u:actual$  afectan el valor de  $u:e$ .

Debemos estar seguros de que cuando se llama PUSH o RELABEL por DISCHARGE, el aplica la operación. El siguiente lema demuestra este hecho.

#### Lema 26.29 Si

DESCARGA llama PUSH.u; / en la línea 7, entonces se aplica una operación de inserción a  $.u; /$ . Si DISCHARGE llama a RELABEL.u/ en la línea 4, entonces se aplica una operación de reetiquetado a  $u$ .

Prueba Las pruebas en las líneas 1 y 6 aseguran que una operación de empuje ocurre solo si se aplica la operación, lo que prueba la primera declaración en el lema.

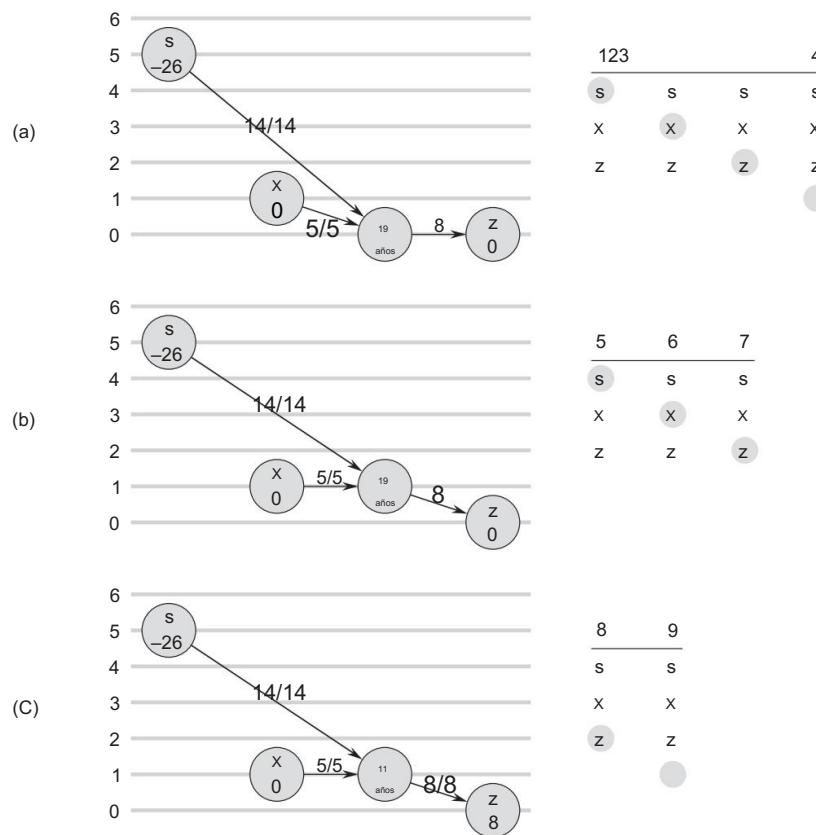


Figura 26.9 Descarga de un vértice  $y$ . Se necesitan 15 iteraciones del ciclo while de DESCARGA para empujar todo el exceso de flujo de  $y$ . Solo se muestran los vecinos de  $y$  y los bordes de la red de flujo que entran o salen de  $y$ . En cada parte de la figura, el número dentro de cada vértice es su exceso al comienzo de la primera iteración que se muestra en la parte, y cada vértice se muestra en su altura en toda la parte. La lista de vecinos  $y:N$  al comienzo de cada iteración aparece a la derecha, con el número de iteración en la parte superior. El vecino sombreado es  $y:\text{actual}$ . (a) Inicialmente, hay 19 unidades de exceso para empujar desde  $y$ , y  $y$ :  $s$  actual . Las iteraciones 1, 2 y 3 solo avanzan  $y:\text{actual}$ , ya que no hay aristas admisibles que salgan de  $y$ . En la iteración 4,  $y:\text{current} \leftarrow \text{NIL}$  (mostrado por el sombreado debajo de la lista de vecinos), y así se vuelve a etiquetar  $y$ :  $y:\text{current}$  se restablece al encabezado de la lista de vecinos. (b) Después de reetiquetar, el vértice  $y$  tiene una altura de 1. En las iteraciones 5 y 6, las aristas  $.y / s$  y  $.y / z$ ; Se encuentra que  $x$  es inadmisible, pero la iteración 7 empuja 8 unidades de exceso de flujo de  $y$  a  $z$ . Debido al impulso,  $y:\text{current}$  no avanza en esta iteración. (c) Debido a que el impulso en la iteración 7 saturó el borde  $.y / z$ , se encuentra inadmisible en la iteración 8. En la iteración 9,  $y:\text{current} \leftarrow \text{NIL}$ , por lo que se vuelve a etiquetar el vértice  $y$  y se restablece

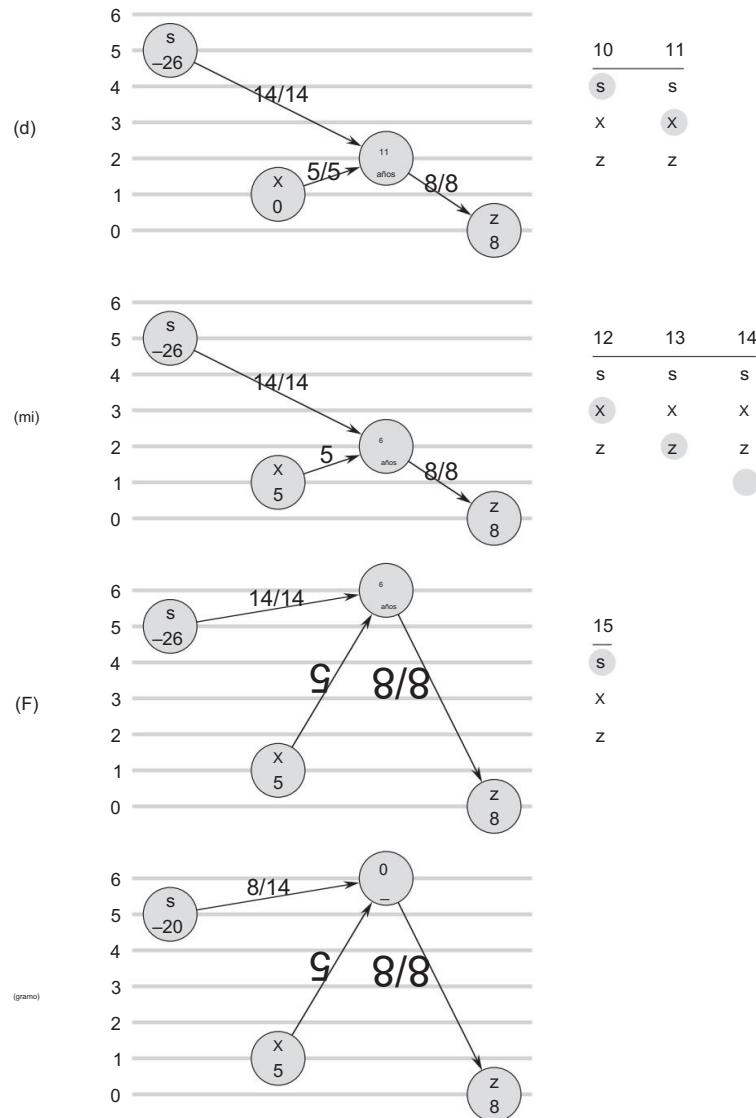


Figura 26.9, continuación (d) En la iteración 10, .y/ s/ es inadmisible, pero la iteración 11 empuja 5 unidades de exceso de flujo de y a x. (e) Debido a que y:current no avanzó en la iteración 11, la iteración 12 encuentra .y/ x/ ser inadmissible. La iteración 13 encuentra .y/ / inadmissible, y la iteración 14 vuelve a etiquetar el vértice y y restablece y:actual. (f) La iteración 15 empuja 6 unidades de exceso de flujo de y a s. (g) El vértice y ahora no tiene exceso de flujo y la DESCARGA termina. En este ejemplo, DISCHARGE comienza y termina con el puntero actual al principio de la lista de vecinos, pero en general no es necesario que sea el caso.

Para probar la segunda afirmación, de acuerdo con la prueba de la línea 1 y el Lema 26.28, solo necesitamos demostrar que todas las aristas que salen de  $u$  son inadmisibles. Si una llamada a  $\text{DISCHARGE}.u/$  comienza con el puntero  $u:\text{current}$  al principio de la lista de vecinos de  $u$  y termina con él al final de la lista, todos los bordes salientes de  $u$  son admisibles y se aplica una operación de reetiquetado. Sin embargo, es posible que durante una llamada a  $\text{DISCHARGE}.u/$ , el puntero  $u:\text{current}$  atraviese solo una parte de la lista antes de que el procedimiento regrese. Entonces pueden ocurrir llamadas a  $\text{DISCHARGE}$  en otros vértices, pero  $u:\text{current}$  continuará moviéndose a través de la lista durante la próxima llamada a  $\text{DISCHARGE}.u/$ . Ahora consideraremos lo que sucede durante un recorrido completo por la lista, que comienza en la cabecera de  $u:N$  y termina con  $u:\text{current} \leftarrow \text{NIL}$ . Una vez que  $u:\text{current}$  llega al final de la lista, el procedimiento vuelve a etiquetar  $u$  y comienza una nueva pasada. Para que el puntero  $u:\text{current}$  avance más allá de un vértice  $2 u:N$  durante un pase, el borde  $.u:/$  debe considerarse inadmisible mediante la prueba de la línea 6. Por lo tanto, cuando se completa el pase, se ha determinado que todos los bordes que salen de  $u$  son inadmisibles en algún momento durante el pase. La observación clave es que al final del pase, cada borde que sale de  $u$  sigue siendo inadmisible. ¿Por qué? Según el Lema 26.27, los empujes no pueden crear ningún borde admisible, independientemente del vértice desde el que se empuje el flujo.

Por lo tanto, cualquier borde admisible debe crearse mediante una operación de reetiquetado. Pero el vértice  $u$  no se vuelve a etiquetar durante el pase, y por el Lema 26.28, cualquier otro vértice que se vuelve a etiquetar durante el pase (como resultado de una llamada de  $\text{DESCARGA}./$ ) no tiene aristas admisibles de entrada después del reetiquetado. Así, al final del pase, todas las aristas que dejan  $u$  quedan inadmisibles, lo que completa la demostración. ■

#### El algoritmo de reetiquetado al frente

En el algoritmo de reetiquetado al frente, mantenemos una lista enlazada  $L$  que consta de todos los vértices en  $V$   $fs;tg$ . Una propiedad clave es que los vértices en  $L$  están ordenados topológicamente de acuerdo con la red admisible, como veremos en el bucle invariante que sigue. (Recuerde del Lema 26.26 que la red admisible es un dag.)

El pseudocódigo para el algoritmo de reetiquetado al frente asume que las listas de vecinos  $u:N$  ya han sido creadas para cada vértice  $u$ . También supone que  $u:\text{next}$  apunta al vértice que sigue a  $u$  en la lista  $L$  y que, como siempre,  $u:\text{next} \leftarrow \text{NIL}$  si  $u$  es el último vértice de la lista.

```

REMOVER LA ETIQUETA AL FRENTE.G; s; t/
1 INICIALIZAR-PREFLUJO.G; s/ 2
LDG:V fs;tg, en cualquier orden 3 para
cada vértice u 2 G:V fs;tg 4 u:actual D
u:N:head 5 u D L:head 6 while u ≠ NIL
7 old-height D u:h 8
DESCARGA.u/ si u:h
> altura antigua mueve u al
frente de la lista L
9
10
11      tu D tu:siguiente

```

El algoritmo de reetiquetado al frente funciona de la siguiente manera. La línea 1 inicializa el preflujo y las alturas a los mismos valores que en el algoritmo genérico push-relabel. La línea 2 inicializa la lista L para que contenga todos los vértices potencialmente desbordados, en cualquier orden. Las líneas 3 y 4 inicializan el puntero actual de cada vértice u en el primer vértice de la lista de vecinos de u.

Como ilustra la figura 26.10, el ciclo while de las líneas 6 a 11 recorre la lista L, descargando vértices. La línea 5 hace que comience con el primer vértice de la lista. Cada vez que pasa por el bucle, la línea 8 descarga un vértice u. Si u fue reetiquetado por el procedimiento DESCARGA , la línea 10 lo mueve al frente de la lista L. Podemos determinar si u fue reetiquetado comparando su altura antes de la operación de descarga, guardada en la variable old-height en la línea 7, con su altura después, en la línea 9.

La línea 11 hace que la próxima iteración del ciclo while use el vértice que sigue a u en la lista L. Si la línea 10 movió u al frente de la lista, el vértice usado en la siguiente iteración es el que sigue a u en su nueva posición en la lista .

Para mostrar que RELABEL-TO-FRONT calcula un flujo máximo, mostraremos que es una implementación del algoritmo genérico push-relabel. Primero, observe que realiza operaciones de inserción y reetiquetado solo cuando se aplican, ya que el Lema 26.29 garantiza que DISCHARGE las realiza solo cuando se aplican.

Queda por mostrar que cuando termina RELABEL-TO-FRONT , no se aplican operaciones básicas. El resto del argumento de corrección se basa en el siguiente ciclo invariante:

En cada prueba en la línea 6 de RELABEL-TO-FRONT, la lista L es una ordenación topológica de los vértices en la red admisible  $G_f;h$  D .V;  $E_f;h/$ , y ningún vértice antes de u en la lista tiene exceso de flujo.

Inicialización: Inmediatamente después de ejecutar INITIALIZE-PREFLOW ,  $s:h$  D  $jV$  j y  $:h$  D 0 para todos los 2 V fsg. Como  $jV$  j 2 (porque V contiene en

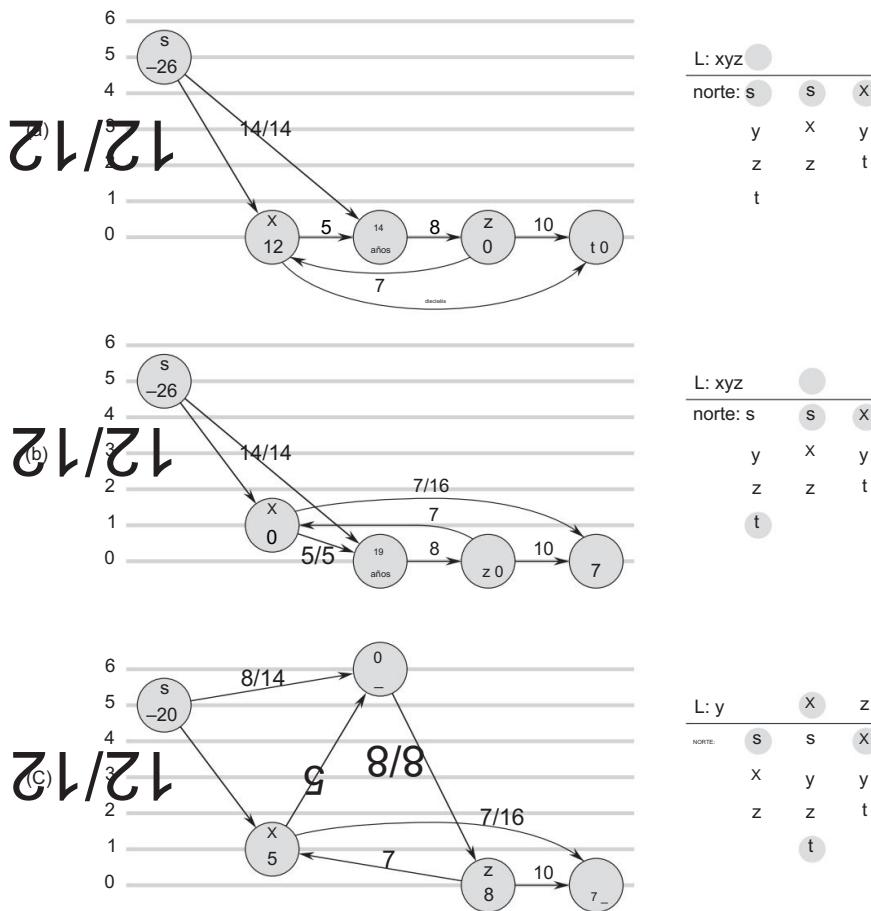


Figura 26.10 La acción de RELABEL-TO-FRONT. (a) Una red de flujo justo antes de la primera iteración del ciclo while . Inicialmente, 26 unidades de caudal salen de la fuente  $s$ . A la derecha se muestra la lista inicial  $LD_{hx}$ ;  $y$ ;  $i$ , donde inicialmente  $u \neq x$ . Debajo de cada vértice en la lista  $L$  está su lista de vecinos, con el vecino actual sombreado. El vértice  $x$  se descarga. Se vuelve a etiquetar a la altura 1, 5 unidades de exceso de flujo se empujan a  $y$ , y las 7 unidades restantes de exceso se empujan al sumidero  $t$ . Debido a que  $x$  se vuelve a etiquetar, se mueve a la cabeza de  $L$ , lo que en este caso no cambia la estructura de  $L$ . (b) Despues de  $x$ , el siguiente vértice en  $L$  que se descarga es  $y$ . La figura 26.9 muestra la acción detallada de descargar  $y$  en esta situación. Debido a que  $y$  se vuelve a etiquetar, se mueve a la cabeza de  $L$ . (c) El vértice  $x$  ahora sigue a  $y$  en  $L$ , por lo que se descarga nuevamente, empujando las 5 unidades de exceso de flujo hacia  $t$ . Debido a que el vértice  $x$  no se vuelve a etiquetar en esta operación de descarga, permanece en su lugar en la lista  $L$ .

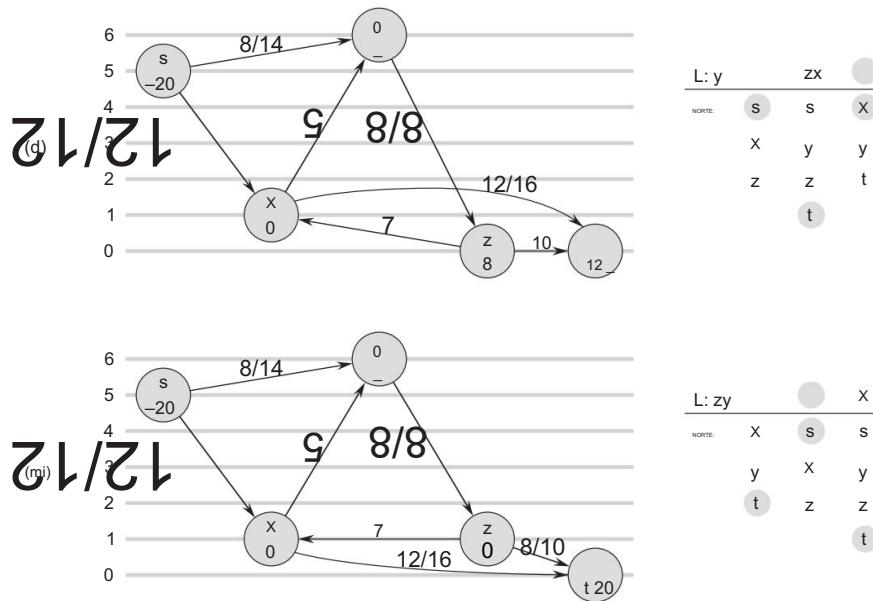


Figura 26.10, continuación (d) Dado que el vértice ' sigue al vértice x en L, se descarga. Se vuelve a etiquetar a la altura 1 y las 8 unidades de exceso de flujo se empujan a t. Debido a que ' se vuelve a etiquetar, se mueve al frente de L. (e) El vértice y ahora sigue al vértice ' en L y, por lo tanto, se descarga. Pero debido a que y no tiene exceso, DESCARGA regresa inmediatamente, y permanece en su lugar en L. El vértice x entonces se descarga. Debido a que tampoco tiene exceso, DESCARGA nuevamente regresa y x permanece en su lugar en L. REETIQUETAR AL FRENTE ha llegado al final de la lista L y termina. No hay vértices desbordantes y el preflujo es un flujo máximo.

menos s y t), ningún borde puede ser admisible. Así,  $Ef;h \leq D$ ; y cualquier ordenación de  $V fs;tg$  es una especie topológica de  $Gf;h$ .

Debido a que u es inicialmente la cabeza de la lista L, no hay vértices antes de ella y, por lo tanto, no hay ninguno antes con exceso de flujo.

Mantenimiento: para ver que cada iteración del ciclo while mantiene el ordenamiento topológico, comenzamos observando que la red admisible se cambia solo mediante operaciones de inserción y reetiquetado. Por el Lema 26.27, las operaciones de empuje no hacen que los bordes se vuelvan admisibles. Por lo tanto, solo las operaciones de reetiquetado pueden crear bordes admisibles. Sin embargo, después de que se vuelve a etiquetar un vértice u, el Lema 26.28 establece que no hay aristas admisibles que entran en u, pero puede haber aristas admisibles que salgan de u. Por lo tanto, al mover u al frente de L, el algoritmo asegura que cualquier borde admisible que deje u satisfaga el ordenamiento topológico.

Para ver que ningún vértice que precede a  $u$  en  $L$  tiene exceso de flujo, denotamos el vértice que será  $u$  en la próxima iteración por  $u_0$ . Los vértices que precederán a  $u_0$  en la siguiente iteración incluyen la  $u$  actual (debido a la línea 11) y ningún otro vértice (si  $u$  se vuelve a etiquetar) o los mismos vértices que antes (si  $u$  no se vuelve a etiquetar). Cuando  $u$  se descarga, no tiene exceso de flujo después. Por lo tanto, si se vuelve a etiquetar  $u$  durante la descarga, ningún vértice anterior a  $u_0$  tiene exceso de flujo. Si  $u$  no se vuelve a etiquetar durante la descarga, ningún vértice anterior a él en la lista adquirió exceso de flujo durante esta descarga, porque  $L$  permaneció ordenado topológicamente en todo momento durante la descarga (como se acaba de señalar, los bordes admisibles se crean solo al volver a etiquetar, no empujar), por lo que cada operación de empuje hace que el exceso de flujo se mueva solo a los vértices más abajo en la lista ( $o$  a  $s$  o  $t$ ). Nuevamente, ningún vértice anterior a  $u_0$  tiene exceso de flujo.

Terminación: cuando el ciclo termina,  $u$  está justo después del final de  $L$ , por lo que el ciclo invariante asegura que el exceso de cada vértice sea 0. Por lo tanto, no se aplican operaciones básicas.

### Análisis

Ahora mostraremos que RELABEL-TO-FRONT se ejecuta en tiempo  $O(V^3)$  en cualquier red de flujo  $G = (V, E)$ . Dado que el algoritmo es una implementación del algoritmo genérico push-relabel, aprovecharemos el corolario 26.21, que proporciona un límite  $O(V^2)$  sobre el número de operaciones de reetiquetado ejecutadas por vértice y un límite  $O(V^3)$  sobre el número total de operaciones de reetiquetado. Además, el ejercicio 26.4-3 proporciona un límite  $O(VE)$  sobre el tiempo total dedicado a realizar operaciones de reetiquetado, y el Lema 26.22 proporciona un límite  $O(VE)$  sobre el número total de operaciones de empuje de saturación.

### Teorema 26.30

El tiempo de ejecución de RELABEL-TO-FRONT en cualquier red de flujo  $G = (V, E)$  es  $O(V^3)$ .

**Prueba** Consideremos una "fase" del algoritmo de reetiquetado al frente como el tiempo entre dos operaciones de reetiquetado consecutivas. Hay fases  $O(V^2)$ , ya que hay operaciones de reetiquetado  $O(V^2)$ . Cada fase consta como máximo de  $jV$  llamadas a DISCHARGE, que podemos ver a continuación. Si DISCHARGE no realiza una operación de reetiquetado, la siguiente llamada a DISCHARGE está más abajo en la lista  $L$ , y la longitud de  $L$  es menor que  $jV$ . Si DISCHARGE realiza un reetiquetado, la siguiente llamada a DISCHARGE pertenece a una fase diferente. Dado que cada fase contiene como máximo  $jV$  llamadas a DESCARGA y hay fases  $O(V^2)$ , el número de veces que se llama DESCARGA en la línea 8 de RELABEL-TO-FRONT es  $O(V^3)$ . Así, el total

el trabajo realizado por el ciclo while en RELABEL-TO-FRONT, excluyendo el trabajo realizado dentro de DISCHARGE, es como máximo OV 3/.

Ahora debemos vincular el trabajo realizado dentro de DISCHARGE durante la ejecución del algoritmo. Cada iteración del ciclo while dentro de DISCHARGE realiza una de tres acciones. Analizaremos la cantidad total de trabajo involucrado en la realización de cada una de estas acciones.

Comenzamos con las operaciones de reetiquetado (líneas 4 y 5). El ejercicio 26.4-3 proporciona un límite de tiempo de O.VE/ en todos los reetiquetados de OV 2/ que se realizan.

Ahora, suponga que la acción actualiza el puntero  $u:\text{actual}$  en la línea 8. Esta acción ocurre  $O.\text{grado}.u//$  veces cada vez que se vuelve a etiquetar un vértice  $u$ , y  $OV \text{ grado}.u//$  veces en general para el vértice. Por lo tanto, para todos los vértices, la cantidad total de trabajo realizado al hacer avanzar los punteros en las listas de vecinos es O.VE/ según el lema del apretón de manos (ejercicio B.4-1).

El tercer tipo de acción realizada por DISCHARGE es una operación de empuje (línea 7). Ya sabemos que el número total de operaciones de empuje de saturación es O.VE/. Obsérvese que si se ejecuta una pulsación no saturada, DESCARGA regresa inmediatamente, ya que la pulsación reduce el exceso a 0. Así, puede haber como máximo una pulsación no saturada por llamada a DESCARGA. Como hemos observado, DESCARGA se llama OV 3/ veces y, por lo tanto, el tiempo total empleado en realizar impulsos no saturados es OV 3/.

El tiempo de ejecución de RELABEL-TO-FRONT es por lo tanto OV es <sup>3</sup> CVE/, que OV 3/. ■

## Ejercicios

### 26.5-1

Ilustre la ejecución de RELABEL-TO-FRONT a la manera de la Figura 26.10 para la red de flujo de la Figura 26.1(a). Suponga que el orden inicial de los vértices en L es  $h1; 2; 3; 4i$  y que las listas de vecinos son

- 1:N D hora; 2; 3i;
- 2:N D hora; 1; 3; 4i;
- 3:ND  $h1 ; 2; 4; ti;$
- 4:ND  $h2 ; 3; ti;$

### 26.5-2 ?

Nos gustaría implementar un algoritmo push-relabel en el que mantenemos una cola de vértices desbordados de primero en entrar, primero en salir. El algoritmo descarga repetidamente el vértice al principio de la cola, y los vértices que no se desbordaban antes de la descarga pero se desbordan después se colocan al final de la cola.

Después de que se descarga el vértice en la cabeza de la cola, se elimina. Cuando el

la cola está vacía, el algoritmo termina. Muestre cómo implementar este algoritmo para calcular un flujo máximo en OV 3/tiempo.

#### 26.5-3

Muestre que el algoritmo genérico aún funciona si RELABEL actualiza  $u:h$  simplemente calculando  $u:h \leftarrow h - C_1$ . ¿Cómo afectaría este cambio al análisis de RELABEL-TO-FRONT?

#### 26.5-4 ?

Muestre que si siempre descargamos un vértice de desbordamiento más alto, podemos hacer que el método push-relabel se ejecute en OV 3/tiempo.

#### 26.5-5

Suponga que en algún punto de la ejecución de un algoritmo push-relabel, existe un entero  $0 < k \leq j \leq V - 1$  para el cual ningún vértice tiene  $:h \geq k$ . Muestre que todos los vértices con  $:h > k$  están en el lado de la fuente de un corte mínimo. Si tal  $k$  existe, la heurística de brecha actualiza cada vértice  $V \setminus f_{\text{sg}}$  para el cual  $:h > k$ , para establecer  $:h \leftarrow \max(:h, jV - j + 1)$ . Muestre que el atributo resultante  $h$  es una función de altura.

(La heurística de brecha es crucial para hacer que las implementaciones del método push-relabel funcionen bien en la práctica).

## Problemas

### 26-1 Problema de escape

Una cuadrícula de  $n \times n$  es un gráfico no dirigido que consta de  $n$  filas y  $n$  columnas de vértices, como se muestra en la figura 26.11. Denotamos el vértice en la  $i$ -ésima fila y la  $j$ -ésima columna por  $(i, j)$ . Todos los vértices de una cuadrícula tienen exactamente cuatro vecinos, excepto los vértices de los límites, que son los puntos  $(i, j)$  para el cual  $i = 1, i = n, j = 1$  o  $j = n$ .

Dado  $m \leq n^2$  puntos de partida  $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$  en la cuadrícula, el problema de escape consiste en determinar si hay o no  $m$  caminos disjuntos de vértice desde los puntos iniciales hasta cualquier  $n$  puntos diferentes en el límite. Por ejemplo, la cuadrícula de la figura 26.11(a) tiene un escape, pero la cuadrícula de la figura 26.11(b) no.

a. Considere una red de flujo en la que los vértices, así como los bordes, tienen capacidades.

Es decir, el flujo positivo total que ingresa a cualquier vértice dado está sujeto a una restricción de capacidad. Demuestre que la determinación del flujo máximo en una red con capacidades de borde y vértice se puede reducir a un problema ordinario de flujo máximo en una red de flujo de tamaño comparable.

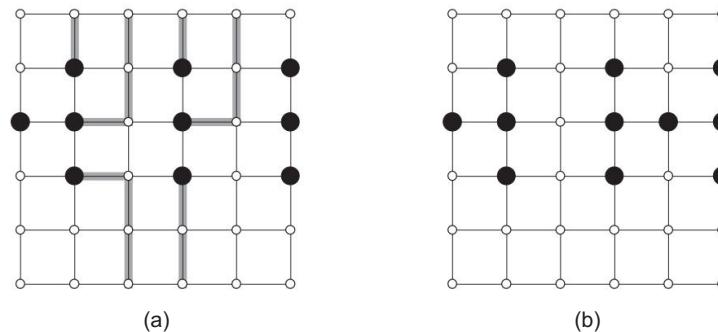


Figura 26.11 Cuadrículas para el problema de escape. Los puntos de inicio son negros y los demás vértices de la cuadrícula son blancos. (a) Una cuadrícula con un escape, mostrada por caminos sombreados. (b) Una cuadrícula sin escape.

- b. Describir un algoritmo eficiente para resolver el problema de escape y analizar su tiempo de ejecución.

## 26-2 Cobertura de trayecto mínima

Cobertura de trayecto de un grafo dirigido  $G$ .  $V$ ;  $E$  es un conjunto  $P$  de caminos disjuntos de vértice tal que cada vértice en  $V$  está incluido exactamente en un camino en  $P$ . Los caminos pueden comenzar y terminar en cualquier lugar, y pueden tener cualquier longitud, incluido 0. Una cobertura de camino mínima de  $G$  es una cubierta de ruta que contiene la menor cantidad posible de rutas.

- a. Proporcione un algoritmo eficiente para encontrar un recorrido mínimo de un grafo acíclico dirigido  $G_D$ .V; M/. (Sugerencia: suponiendo que VD f1; 2; ; ; ; ng, construya el gráfico G0 D .V E0 /, donde

$\nabla^0 = Df(x_0) : x_1 \mapsto x_{n+1}$  [  $f'(x_0) : y_1 \mapsto y_{n+1}$  ]

$$E_0 \oplus f \times 0 : v_i / W_{i-2} V \cong [f(v_i) \times 0] / W_{i-2} V \cong [f(v_i) \times v_i] / W_{i-2} V / 0 = E_0$$

y ejecutar un algoritmo de flujo máximo.

- b. : Su algoritmo funciona para gráficos dirigidos que contienen ciclos? Explicar.

### 26.3 Consultoría algorítmica El profesor

Gore quiere abrir una empresa de consultoría algorítmica. Ha identificado n subáreas importantes de algoritmos (correspondientes aproximadamente a diferentes partes de este libro de texto), que representa mediante el conjunto  $AD = \{A_1, A_2, \dots, A_n\}$ . En cada subárea  $A_k$ , puede contratar a un experto en esa área por  $c_k$  dólares. La consultora ha alineado un conjunto  $JD = \{J_1, J_2, \dots, J_m\}$  de trabajos potenciales. Para realizar el trabajo  $J_i$ , la empresa necesita haber contratado expertos en un subconjunto  $R_i \subseteq AD$ .

subáreas. Cada experto puede trabajar en múltiples trabajos simultáneamente. Si la empresa elige aceptar el trabajo  $J_i$ , debe haber contratado expertos en todas las subáreas en  $R_i$ , y obtendrá ingresos de dólares  $p_i$ .

El trabajo del profesor Gore es determinar en qué subáreas contratar expertos y qué trabajos aceptar para maximizar los ingresos netos, que son los ingresos totales de los trabajos aceptados menos el costo total de emplear a los expertos.

Considere la siguiente red de flujo  $G$ . Contiene un vértice fuente  $s$ , vértices  $A_1; A_2; \dots; A_n$ , vértices  $J_1; J_2; \dots; J_m$ , y un vértice hundido  $t$ . Para  $k \in D(1; 2; \dots; n)$ , la red de flujo contiene un borde  $.s; A_k/$  con capacidad  $c_{sA_k}$ ;  $A_k/ D$   $c_k$ , y para  $i \in D(1; 2; \dots; m)$ , la red de flujo contiene un borde  $.J_i; t/$  con capacidad  $c_{J_i t}$ ;  $t/ D p_i$ . Para  $k \in D(1; 2; \dots; n)$  y  $i \in D(1; 2; \dots; m)$ , si  $A_k \in R_i$ , entonces  $G$  contiene una arista  $.A_k; J_i/$  con capacidad  $c_{A_k J_i}$ ;  $J_i/ D 1$ .

- Demuestre que si  $J_i \geq T$  para un corte de capacidad finita  $.S; T/$  de  $G$ , luego  $A_k \geq T$  para cada  $A_k \in R_i$ .
- Muestre cómo determinar el ingreso neto máximo de la capacidad de un corte mínimo de  $G$  y los valores  $p_i$  dados.
- Proporcione un algoritmo eficiente para determinar qué trabajos aceptar y qué expertos contratar.  
Analice el tiempo de ejecución de su algoritmo en términos de  $m, n$  y  $r D P_m$   
 $\sum_{i=1}^m \sum_{j=1}^{r_i} c_{ij}$ .

#### 26-4 Actualización del caudal máximo

Sea  $G = (V, E)$  sea una red de flujo con fuente  $s$ , sumidero  $t$  y capacidades enteras.

Supongamos que se nos da un caudal máximo en  $G$ .

- Supongamos que aumentamos la capacidad de un solo borde  $.u; / 2 E$  por 1. Proporcione un algoritmo OV CE-time para actualizar el flujo máximo.
- Supongamos que disminuimos la capacidad de un solo borde  $.u; / 2 E$  por 1. Proporcione un algoritmo OV CE-time para actualizar el flujo máximo.

#### 26-5 Caudal máximo por escalado

Sea  $G = (V, E)$  sea una red de flujo con fuente  $s$ , sumidero  $t$  y una capacidad entera  $c_{uv}$  en cada borde  $.u; / 2 E$ . Let  $C = \max_{u,v} c_{uv}$ .

- Argumente que un corte mínimo de  $G$  tiene capacidad como máximo  $C$ .
- Para un número  $K$  dado, muestre cómo encontrar una trayectoria creciente de capacidad en menos  $K$  en  $O(E)$  tiempo, si tal camino existe.

Podemos usar la siguiente modificación del MÉTODO FORD-FULKERSON para calcular un caudal máximo en G:

MAX-FLUJO-ESCALANDO.G; s; t / 1 CD

máx.u;/2E cu; / 2 inicializar flujo f  
a 0 3 KD 2blgCc

4 mientras que K 1

5        mientras exista un camino de aumento p de capacidad al menos K aumenta  
              el flujo f a lo largo de p 6  
7 KDK=2 8 volver f

C. Argumente que MAX-FLOW-BY-SCALING devuelve un flujo máximo.

d. Demostrar que la capacidad de un corte mínimo de la red residual Gf es como máximo  $2K \cdot j_E$   
cada vez que se ejecuta la línea 4.

mi. Argumente que el ciclo while interno de las líneas 5 y 6 ejecuta  $O(E)$  veces para cada valor  
de k

F. Concluya que MAX-FLOW-BY-SCALING puede implementarse para que se ejecute  
en  $O(E \cdot \lg C)$  tiempo.

26-6 El algoritmo de emparejamiento bipartito de Hopcroft-Karp En este problema, describimos un algoritmo más rápido, debido a Hopcroft y Karp, para encontrar un emparejamiento máximo en un grafo bipartito. El algoritmo se ejecuta en tiempo  $O(\sqrt{V} E)$ . Dado un grafo bipartito no dirigido  $G = (V, E)$ , donde  $V = L \cup R$  y todas las aristas tienen exactamente un extremo en  $L$ , sea  $M$  un emparejamiento en  $G$ . Decimos que un camino simple  $P$  en  $G$  es un camino creciente con respecto a  $M$  si comienza en un vértice no emparejado en  $L$ , termina en un vértice no emparejado en  $R$ , y sus aristas pertenecen alternativamente a  $M$  y  $E \setminus M$ . (Esta definición de una ruta de aumento está relacionada con, pero es diferente de, una ruta de aumento en una red de flujo). En este problema, trate un camino como una secuencia de aristas, en lugar de una secuencia de vértices. Un camino de aumento más corto con respecto a una coincidencia  $M$  es un camino de aumento con un número mínimo de aristas.

Dados dos conjuntos A y B, la diferencia simétrica  $A \Delta B$  se define como  $|A \cup B| - |A \cap B|$ , es decir, los elementos que están exactamente en uno de los dos conjuntos.

- a. Muestre que si  $M$  es un emparejamiento y  $P$  es un camino creciente con respecto a  $M$ , entonces la diferencia simétrica  $M \setminus P$  es un emparejamiento y  $|M| - |P| \leq |M| - k$ .
- Demuestre que si  $P_1; P_2; \dots; P_k$  son caminos de aumento disjuntos de vértice con respecto a  $M$ , entonces la diferencia simétrica  $M \setminus (P_1 \cup P_2 \cup \dots \cup P_k)$  es una correspondencia con cardinalidad  $|M| - k$ .

La estructura general de nuestro algoritmo es la siguiente:

HOPCROFT-KARP.G/ 1

MD ; 2 repetir

3 dejar PD

f $P_1; P_2; \dots; P_k$  sea un conjunto máximo de caminos de aumento más cortos  
disjuntos de vértice con respecto a  $M$

4 MDM  $\setminus (P_1 \cup P_2 \cup \dots \cup P_k)$  hasta  $P ==$  ; 6 vuelta M

El resto de este problema le pide que analice el número de iteraciones en el algoritmo (es decir, el número de iteraciones en el ciclo de repetición) y que describa una implementación de la línea 3.

- b. Dadas dos coincidencias  $M$  y  $M'$  en  $G$ , demuestre que cada vértice en el gráfico  $G \setminus (M \cup M')$  tiene grado a lo sumo 2. Concluya que  $G \setminus (M \cup M')$  es una unión disjunta de caminos o ciclos simples. Argumente que las aristas en cada camino simple o ciclo pertenecen alternativamente a  $M$  o  $M'$ . Demuestre que si  $|M| < |M'|$ , entonces  $M \setminus M'$  contiene al menos  $|M| - |M'|$  caminos de aumento disjuntos de vértice con respecto a  $M$ .

Sea  $l$  la longitud de un camino creciente más corto con respecto a un  $M$  coincidente, y sea  $P_1; P_2; \dots; P_k$  sea un conjunto máximo de caminos crecientes disjuntos de vértice de longitud  $l$  con respecto a  $M$ . Sea  $M' = M \setminus (P_1 \cup P_2 \cup \dots \cup P_k)$ , y suponga que  $P$  es un camino creciente más corto con respecto a  $M'$ .

C. Muestre que si  $P$  es disjunto de vértice de  $P_1; P_2; \dots; P_k$ , entonces  $P$  tiene más de  $l$  bordes

d. Supongamos ahora que  $P$  no es un vértice disjunto de  $P_1; P_2; \dots; P_k$ . Sea  $A$  el conjunto de aristas  $(M \setminus M') \cap P$ . Demuestre que  $|A| \geq l$ . Concluya que  $P$  tiene más de  $l$  aristas. que  $|A|$

mi. Demuestre que si un camino de aumento más corto con respecto a  $M$  tiene  $l$  aristas, el tamaño de la coincidencia máxima es como máximo  $|M| - l$ .

F. Demuestre que el número de iteraciones repetidas del bucle en el algoritmo es como máximo  $2 \sum_{j=1}^M p_j V_j$ .  
 (Pista: ¿Cuánto puede crecer  $M$  después del número de iteración  $p_j V_j$ ?)

gramo. Proporcione un algoritmo que se ejecute en  $O(E/tiempo)$  para encontrar un conjunto máximo de caminos de aumento más cortos disjuntos de vértice  $P_1; P_2; \dots; P_k$  para un  $M$  coincidente determinado. Concluya que el tiempo total de ejecución de HOPCROFT-KARP es  $O(pV E)$ .

### Notas del capítulo

Ahuja, Magnanti y Orlin [7], Even [103], Lawler [224], Papadimitriou y Steiglitz [271] y Tarjan [330] son buenas referencias para el flujo de red y los algoritmos relacionados. Goldberg, Tardos y Tarjan [139] también brindan un buen estudio de algoritmos para problemas de flujo de redes, y Schrijver [304] ha escrito una revisión interesante de los desarrollos históricos en el campo de los flujos de redes.

El método de Ford-Fulkerson se debe a Ford y Fulkerson [109], quienes originaron el estudio formal de muchos de los problemas en el área del flujo de red, incluidos los problemas de flujo máximo y emparejamiento bipartito. Muchas implementaciones tempranas del método Ford-Fulkerson encontraron rutas de aumento utilizando la búsqueda primero en amplitud; Edmonds y Karp [102], e independientemente Dinic [89], demostraron que esta estrategia produce un algoritmo de tiempo polinomial. Dinic [89] también desarrolló por primera vez una idea relacionada, la de usar "flujos de bloqueo". Karzanov [202] primero desarrolló la idea de preflujos. El método push-relabel se debe a Goldberg [136] y Goldberg y Tarjan [140]. Goldberg y Tarjan dieron un algoritmo  $O(V E \lg V)^2 = O(E^2 \lg V)$  que usa una cola para mantener el conjunto de vértices desbordados, así como un algoritmo que usa árboles dinámicos para lograr un tiempo de ejecución de  $O(V E \lg V)$ . Varios otros investigadores han desarrollado algoritmos de flujo máximo push-relabel. Ahuja y Orlin [9] y Ahuja, Orlin y Tarjan [10] proporcionaron algoritmos que usaban escalado. Cheriyan y Maheshwari [62] propusieron empujar el flujo desde el vértice desbordante de altura máxima. Cheriyan y Hagerup [61] sugirieron permutar aleatoriamente las listas de vecinos, y varios investigadores [14, 204, 276] desarrollaron ingeniosas des-aleatorizaciones de esta idea, lo que llevó a una secuencia de algoritmos más rápidos. El algoritmo de King, Rao y Tarjan [204] es el algoritmo más rápido y se ejecuta en  $O(V E \log E) = O(V E \lg V)$ .

El algoritmo asintóticamente más rápido hasta la fecha para el problema de flujo máximo, de Goldberg y Rao [138], se ejecuta en el tiempo  $O(\min\{V^2, E^2\})$ ;  $E^2 = O(V E \lg V)$ , donde  $C = \max\{u_i, v_i\}$ ;  $\ell = \min\{u_i, v_i\}$ . Este algoritmo no utiliza el método push-relabel, sino que se basa en encontrar flujos de bloqueo. Todos los algoritmos de flujo máximo anteriores, incluidos los de este capítulo, usan alguna noción de distancia (los algoritmos push-relabel usan la noción análoga de altura), con una longitud de 1

asignado implícitamente a cada borde. Este nuevo algoritmo adopta un enfoque diferente y asigna una longitud de 0 a los bordes de alta capacidad y una longitud de 1 a los bordes de baja capacidad.

De manera informal, con respecto a estas longitudes, las rutas más cortas desde la fuente hasta el sumidero tienden a tener una capacidad alta, lo que significa que se deben realizar menos iteraciones.

En la práctica, los algoritmos push-relabel actualmente dominan los algoritmos basados en programación lineal o de ruta aumentada para el problema de flujo máximo. Un estudio de Cherkassky y Goldberg [63] subraya la importancia de utilizar dos heurísticas al implementar un algoritmo push-relabel. La primera heurística consiste en realizar periódicamente una búsqueda en anchura de la red residual para obtener valores de altura más precisos. La segunda heurística es la heurística de la brecha, descrita en el ejercicio 26.5-5. Cherkassky y Goldberg concluyen que la mejor opción de las variantes de reetiquetado de empuje es la que elige descargar el vértice desbordante con la altura máxima.

El mejor algoritmo hasta la fecha para la coincidencia bipartita máxima, descubierto por Hopcroft y Karp [176], se ejecuta en tiempo  $O(\sqrt{V} E)$  y se describe en el problema 26-6.

El libro de Lovász y Plummer [239] es una excelente referencia sobre problemas de emparejamiento.



---

## VII Temas Seleccionados

---

## Introducción

Esta parte contiene una selección de temas algorítmicos que amplían y complementan el material anterior de este libro. Algunos capítulos introducen nuevos modelos de computación, como circuitos o computadoras paralelas. Otros cubren dominios especializados como la geometría computacional o la teoría de números. Los últimos dos capítulos analizan algunas de las limitaciones conocidas para el diseño de algoritmos eficientes y presentan técnicas para hacer frente a esas limitaciones.

El Capítulo 27 presenta un modelo algorítmico para computación paralela basado en subprocessos múltiples dinámicos. El capítulo presenta los conceptos básicos del modelo y muestra cómo cuantificar el paralelismo en términos de las medidas de trabajo y amplitud. Luego investiga varios algoritmos de subprocessos múltiples interesantes, incluidos algoritmos para la multiplicación de matrices y la clasificación por fusión.

El Capítulo 28 estudia algoritmos eficientes para operar con matrices. Presenta dos métodos generales, descomposición LU y descomposición LUP, para resolver ecuaciones lineales mediante eliminación gaussiana en tiempo  $O.n^3/$ . También muestra que la inversión de matrices y la multiplicación de matrices se pueden realizar con la misma rapidez. El capítulo concluye mostrando cómo calcular una solución aproximada de mínimos cuadrados cuando un conjunto de ecuaciones lineales no tiene una solución exacta.

El capítulo 29 estudia la programación lineal, en la que deseamos maximizar o minimizar un objetivo, dados los recursos limitados y las restricciones que compiten entre sí. La programación lineal surge en una variedad de áreas de aplicación práctica. Este capítulo cubre cómo formular y resolver programas lineales. El método de solución cubierto es el algoritmo simplex, que es el algoritmo más antiguo para programación lineal. A diferencia de muchos algoritmos de este libro, el algoritmo simplex no se ejecuta en tiempo polinomial en el peor de los casos, pero es bastante eficiente y se usa mucho en la práctica.

El capítulo 30 estudia las operaciones con polinomios y muestra cómo utilizar una técnica de procesamiento de señales bien conocida, la transformada rápida de Fourier (FFT), para multiplicar dos polinomios de grado  $n$  en tiempo  $O(n \lg n)$ . También investiga implementaciones eficientes de la FFT, incluido un circuito paralelo.

El capítulo 31 presenta algoritmos de teoría de números. Después de repasar la teoría de números elemental, presenta el algoritmo de Euclides para calcular los máximos comunes divisores. A continuación, estudia algoritmos para resolver ecuaciones lineales modulares y para elevar un número a un módulo de potencia de otro número. Luego, explora una aplicación importante de los algoritmos teóricos de números: el criptosistema de clave pública RSA.

Este criptosistema se puede utilizar no solo para cifrar mensajes para que un adversario no pueda leerlos, sino también para proporcionar firmas digitales. Luego, el capítulo presenta la prueba de primalidad aleatoria de Miller-Rabin, con la cual podemos encontrar primos grandes de manera eficiente, un requisito esencial para el sistema RSA. Finalmente, el capítulo cubre la heurística "rho" de Pollard para la factorización de enteros y analiza el estado del arte de la factorización de enteros.

El capítulo 32 estudia el problema de encontrar todas las apariciones de una cadena de patrón dada en una cadena de texto dada, un problema que surge con frecuencia en los programas de edición de texto. Después de examinar el enfoque ingenuo, el capítulo presenta un enfoque elegante debido a Rabin y Karp. Luego, después de mostrar una solución eficiente basada en autómatas finitos, el capítulo presenta el algoritmo de Knuth-Morris-Pratt, que modifica el algoritmo basado en autómatas para ahorrar espacio mediante el preprocesamiento inteligente del patrón.

El Capítulo 33 considera algunos problemas de geometría computacional. Después de discutir las primitivas básicas de la geometría computacional, el capítulo muestra cómo usar un método de "barrido" para determinar de manera eficiente si un conjunto de segmentos de línea contiene intersecciones. Dos algoritmos ingeniosos para encontrar el casco convexo de un conjunto de puntos, el escaneo de Graham y la marcha de Jarvis, también ilustran el poder de los métodos de barrido. El capítulo cierra con un algoritmo eficiente para encontrar el par más cercano entre un conjunto dado de puntos en el plano.

El capítulo 34 trata de problemas NP-completos. Muchos problemas computacionales interesantes son NP-completos, pero no se conoce ningún algoritmo de tiempo polinomial para resolver ninguno de ellos. Este capítulo presenta técnicas para determinar cuándo un problema es NP-completo. Se ha demostrado que varios problemas clásicos son NP-completos: determinar si un gráfico tiene un ciclo hamiltoniano, determinar si una fórmula booleana es satisfactoria y determinar si un conjunto dado de números tiene un subconjunto que se suma a un valor objetivo dado. El capítulo también demuestra que el famoso problema del viajante de comercio es NP-completo.

El Capítulo 35 muestra cómo encontrar soluciones aproximadas a problemas NP-completos de manera eficiente mediante el uso de algoritmos de aproximación. Para algunos problemas NP-completos, las soluciones aproximadas que son casi óptimas son bastante fáciles de producir, pero para otros, incluso los mejores algoritmos de aproximación conocidos funcionan progresivamente peor a medida que

el tamaño del problema aumenta. Luego, hay algunos problemas en los que podemos invertir cantidades cada vez mayores de tiempo de cálculo a cambio de soluciones aproximadas cada vez mejores. Este capítulo ilustra estas posibilidades con el problema de cobertura de vértices (versiones no ponderadas y ponderadas), una versión de optimización de la satisfacibilidad de 3-CNF, el problema del viajante de comercio, el problema de cobertura de conjuntos y el problema de suma de subconjuntos.

La gran mayoría de los algoritmos de este libro son algoritmos en serie adecuados para ejecutarse en una computadora monoprocesador en la que solo se ejecuta una instrucción a la vez. En este capítulo, extenderemos nuestro modelo algorítmico para abarcar algoritmos paralelos, que pueden ejecutarse en una computadora multiprocesador que permite la ejecución simultánea de múltiples instrucciones. En particular, exploraremos el modelo elegante de algoritmos dinámicos de subprocessos múltiples, que son susceptibles de diseño y análisis algorítmicos, así como de implementación eficiente en la práctica.

Las computadoras paralelas (computadoras con múltiples unidades de procesamiento) se han vuelto cada vez más comunes y abarcan una amplia gama de precios y rendimiento. Los multiprocesadores de chips de computadoras de escritorio y portátiles relativamente económicos contienen un solo chip de circuito integrado de múltiples núcleos que alberga múltiples "núcleos" de procesamiento, cada uno de los cuales es un procesador completo que puede acceder a una memoria común. En un punto intermedio de precio/rendimiento, se encuentran los clústeres creados a partir de computadoras individuales, a menudo máquinas simples de clase PC, con una red dedicada que las interconecta. Las máquinas más caras son las supercomputadoras, que a menudo usan una combinación de arquitecturas personalizadas y redes personalizadas para ofrecer el mayor rendimiento en términos de instrucciones ejecutadas por segundo.

Las computadoras multiprocesador han existido, de una forma u otra, durante décadas. Aunque la comunidad informática se decidió por el modelo de máquina de acceso aleatorio para la computación en serie al principio de la historia de las ciencias de la computación, ningún modelo único para la computación paralela ha ganado tanta aceptación. Una de las principales razones es que los proveedores no se han puesto de acuerdo sobre un modelo arquitectónico único para computadoras paralelas. Por ejemplo, algunas computadoras paralelas cuentan con memoria compartida, donde cada procesador puede acceder directamente a cualquier ubicación de la memoria. Otras computadoras paralelas emplean memoria distribuida, donde la memoria de cada procesador es privada y se debe enviar un mensaje explícito entre procesadores para que un procesador acceda a la memoria de otro. Sin embargo, con la llegada de la tecnología multinúcleo, cada nueva computadora portátil y de escritorio es ahora una computadora paralela de memoria compartida.

y la tendencia parece ser hacia el multiprocesamiento de memoria compartida. Aunque el tiempo lo dirá, ese es el enfoque que tomaremos en este capítulo.

Un medio común de programar multiprocesadores de chip y otras computadoras paralelas de memoria compartida es mediante el uso de subprocesos estáticos, que proporciona una abstracción de software de "procesadores virtuales" o subprocesos que comparten una memoria común. Cada subproceso mantiene un contador de programa asociado y puede ejecutar código independientemente de los otros subprocesos. El sistema operativo carga un subproceso en un procesador para su ejecución y lo desconecta cuando se necesita ejecutar otro subproceso. Aunque el sistema operativo permite a los programadores crear y destruir subprocesos, estas operaciones son comparativamente lentas. Por lo tanto, para la mayoría de las aplicaciones, los subprocesos persisten durante la duración de un cálculo, razón por la cual los llamamos "estáticos".

Desafortunadamente, programar una computadora paralela de memoria compartida directamente usando subprocesos estáticos es difícil y propenso a errores. Una de las razones es que dividir dinámicamente el trabajo entre los subprocesos para que cada subproceso reciba aproximadamente la misma carga resulta ser una tarea complicada. Para cualquier aplicación, excepto para las más simples, el programador debe usar protocolos de comunicación complejos para implementar un planificador para equilibrar la carga del trabajo. Este estado de cosas ha llevado a la creación de plataformas de concurrencia, que proporcionan una capa de software que coordina, programa y administra los recursos de cómputo paralelo. Algunas plataformas de simultaneidad se crean como bibliotecas de tiempo de ejecución, pero otras proporcionan lenguajes paralelos completos con soporte para compilador y tiempo de ejecución.

### Programación multihilo dinámica

Una clase importante de plataforma de concurrencia es el multiproceso dinámico, que es el modelo que adoptaremos en este capítulo. Los subprocesos múltiples dinámicos permiten a los programadores especificar el paralelismo en las aplicaciones sin preocuparse por los protocolos de comunicación, el equilibrio de carga y otros caprichos de la programación de subprocesos estáticos. La plataforma de concurrencia contiene un planificador, que equilibra la carga del cálculo automáticamente, lo que simplifica enormemente la tarea del programador. Aunque la funcionalidad de los entornos dinámicos de subprocesos múltiples todavía está evolucionando, casi todos admiten dos características: paralelismo anidado y bucles paralelos. El paralelismo anidado permite que se "genere" una subrutina, lo que permite que la persona que llama continúe mientras la subrutina generada calcula su resultado. Un bucle paralelo es como un bucle for ordinario , excepto que las iteraciones del bucle pueden ejecutarse simultáneamente.

Estas dos características forman la base del modelo de subprocesos múltiples dinámicos que estudiaremos en este capítulo. Un aspecto clave de este modelo es que el programador necesita especificar solo el paralelismo lógico dentro de un cómputo y los subprocesos dentro de la programación de la plataforma de concurrencia subyacente y equilibrar la carga del cómputo entre ellos. Investigaremos algoritmos multiproceso escritos para

este modelo, así como también cómo la plataforma de concurrencia subyacente puede programar cálculos de manera eficiente.

Nuestro modelo para subprocessos múltiples dinámicos ofrece varias ventajas importantes:

Es una extensión simple de nuestro modelo de programación en serie. Podemos describir un algoritmo de subprocessos múltiples agregando a nuestro pseudocódigo solo tres palabras clave de "conurrencia": paralelo, generación y sincronización. Además, si eliminamos estas palabras clave de concurrencia del pseudocódigo de subprocessos múltiples, el texto resultante es un pseudocódigo en serie para el mismo problema, que llamamos la "serialización" del algoritmo de subprocessos múltiples.

Proporciona una forma teóricamente clara de cuantificar el paralelismo en función de las nociónes de "trabajo" y "intervalo".

Muchos algoritmos de subprocessos múltiples que involucran paralelismo anidado se derivan naturalmente del paradigma divide y vencerás. Además, así como los algoritmos de divide y vencerás en serie se prestan al análisis mediante la resolución de recurrencias, también lo hacen los algoritmos de subprocessos múltiples.

El modelo es fiel a cómo está evolucionando la práctica de la computación paralela. Un número creciente de plataformas de concurrencia admiten una variante u otra de subprocessos múltiples dinámicos, incluidos Cilk [51, 118], Cilk ++ [71], OpenMP [59], Task Parallel Library [230] y Threading Building Blocks [292].

La Sección 27.1 introduce el modelo dinámico de subprocessos múltiples y presenta las métricas de trabajo, extensión y paralelismo, que usaremos para analizar algoritmos de subprocessos múltiples. La Sección 27.2 investiga cómo multiplicar matrices con subprocessos múltiples, y la Sección 27.3 aborda el problema más difícil de la ordenación por combinación de subprocessos múltiples.

## 27.1 Los fundamentos del subprocessamiento múltiple dinámico

Comenzaremos nuestra exploración de subprocessos múltiples dinámicos usando el ejemplo de calcular números de Fibonacci recursivamente. Recuerde que los números de Fibonacci están definidos por recurrencia (3.22):

```
F0 D 0      :
F1 D 1      :
Fi D Fi1 C Fi2 para i > 2
```

Aquí hay un algoritmo serial simple, recursivo, para calcular el  $n$ -ésimo número de Fibonacci:

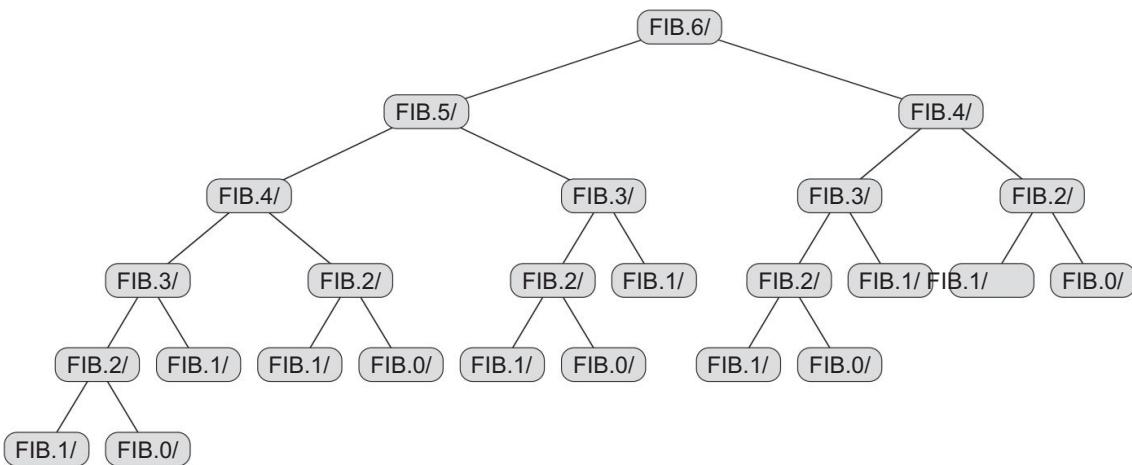


Figura 27.1 El árbol de instancias de procedimientos recursivos al calcular FIB.6/. Cada instancia de FIB con el mismo argumento hace el mismo trabajo para producir el mismo resultado, proporcionando una forma ineficiente pero interesante de calcular los números de Fibonacci.

```

FIB.n/
1 si n 1 2
devuelve n 3 si
no x D FIB.n 1/ 4 y D FIB.n
2/ 5 devuelve x C y
  
```

Realmente no querrías calcular grandes números de Fibonacci de esta manera, porque este cálculo hace mucho trabajo repetido. La figura 27.1 muestra el árbol de instancias de procedimientos recursivos que se crean al calcular F6. Por ejemplo, una llamada a FIB.6/ llama recursivamente a FIB.5/ y luego a FIB.4/. Pero, la llamada a FIB.5/ también resulta en una llamada a FIB.4/. Ambas instancias de FIB.4/ devuelven el mismo resultado (F4 D 3). Dado que el procedimiento FIB no memoriza, la segunda llamada a FIB.4/ replica el trabajo que realiza la primera llamada.

Sea  $T_{.n}$  el tiempo de ejecución de  $FIB.n/$ . Dado que  $FIB.n/$  contiene dos recursivas pasivas más una cantidad constante de trabajo extra, obtenemos la recurrencia

$$T_{.n} = DT_{.n} + CT_{.n} + C_{.1} :$$

Esta recurrencia tiene solución  $T_{.n} = D \cdot F_n + C$ , la cual podemos mostrar usando el método de sustitución. Para una hipótesis inductiva, suponga que  $T_{.n} \geq aF_n + b$ , donde  $a > 1$  y  $b > 0$  son constantes. Sustituyendo, obtenemos

```

T .n/      .aFn1 b/ C .aFn2 b/ C ,.1/
          D a.Fn1 C Fn2/ 2b C ,.1/ D aFn b .b ,.1//  

          aFn b

```

si elegimos b lo suficientemente grande como para dominar la constante en el ,.1/. Entonces podemos elegir un valor lo suficientemente grande como para satisfacer la condición inicial. El límite analítico

T .n/ D ,.n/ ; (27.1)

— donde D .1 C p5/=2 es la proporción áurea, ahora se sigue de la ecuación (3.25).

Dado que Fn crece exponencialmente en n, este procedimiento es una forma particularmente lenta de calcular los números de Fibonacci. (Vea el Problema 31-3 para formas mucho más rápidas).

Aunque el procedimiento FIB es una forma deficiente de calcular los números de Fibonacci, es un buen ejemplo para ilustrar conceptos clave en el análisis de algoritmos de subprocessos múltiples. Observe que dentro de FIB.n/, las dos llamadas recursivas en las líneas 3 y 4 a FIB.n 1/ y FIB.n 2/, respectivamente, son independientes entre sí: se pueden llamar en cualquier orden y el cálculo se realiza por uno de ninguna manera afecta al otro. Por lo tanto, las dos llamadas recursivas pueden ejecutarse en paralelo.

Aumentamos nuestro pseudocódigo para indicar el paralelismo agregando las palabras clave de simultaneidad generar y sincronizar. Así es como podemos reescribir el procedimiento FIB para usar subprocessos múltiples dinámicos:

```

P-FIB.n/ 1
si n 1 2
    volver n 3
más x D generar P-FIB.n 1/ y D P-FIB.n
    2/ 4
5     retorno
6     de sincronización x C y

```

Tenga en cuenta que si eliminamos las palabras clave de concurrencia generadas y sincronizadas de P-FIB, el texto del pseudocódigo resultante es idéntico a FIB (aparte de cambiar el nombre del procedimiento en el encabezado y en las dos llamadas recursivas). Definimos la serialización de un algoritmo de subprocessos múltiples como el algoritmo en serie que resulta de eliminar las palabras clave de subprocessos múltiples: generar, sincronizar y, cuando examinamos bucles paralelos, paralelo. De hecho, nuestro pseudocódigo multiproceso tiene la buena propiedad de que una serialización siempre es un pseudocódigo serial ordinario para resolver el mismo problema.

El paralelismo anidado ocurre cuando la palabra clave spawn precede a una llamada a procedimiento, como en la línea 3. La semántica de un spawn difiere de una llamada a un procedimiento ordinario en que la instancia del procedimiento que ejecuta el spawn (el padre) puede continuar ejecutándose en paralelo con el spawn generado. subrutina, su hijo, en lugar de esperar

para que el niño complete, como sucedería normalmente en una ejecución en serie. En este caso, mientras el hijo generado calcula P-FIB.n 1/, el padre puede pasar a calcular P-FIB.n 2/ en la línea 4 en paralelo con el hijo generado. Dado que el procedimiento P-FIB es recursivo, estas dos subrutinas crean un paralelismo anidado, al igual que sus hijos, creando así un árbol potencialmente enorme de subcálculos, todos ejecutándose en paralelo.

La palabra clave `spawn` no dice, sin embargo, que un procedimiento debe ejecutarse al mismo tiempo que sus hijos generados, solo que puede hacerlo. Las palabras clave de concurrencia expresan el paralelismo lógico del cómputo, indicando qué partes del cómputo pueden proceder en paralelo. En tiempo de ejecución, depende de un programador determinar qué subcálculos se ejecutan realmente al mismo tiempo asignándolos a los procesadores disponibles a medida que se desarrolla el cálculo. Discutiremos la teoría detrás de los programadores en breve.

Un procedimiento no puede usar de forma segura los valores devueltos por sus hijos generados hasta después de ejecutar una declaración de sincronización , como en la línea 5. La palabra clave `sync` indica que el procedimiento debe esperar según sea necesario para que se completen todos sus hijos generados antes de continuar con la declaración posterior. la sincronización En el procedimiento P-FIB, se requiere una sincronización antes de la declaración de devolución en la línea 6 para evitar la anomalía que ocurriría si `xey` se sumaran antes de calcular `x`. Además de la sincronización explícita proporcionada por la declaración de sincronización , cada procedimiento ejecuta una sincronización implícitamente antes de regresar, lo que garantiza que todos sus elementos secundarios finalicen antes que él.

#### Un modelo para la ejecución multiproceso

Es útil pensar en una computación de subprocessos múltiples (el conjunto de instrucciones de tiempo de ejecución ejecutadas por un procesador en nombre de un programa de subprocessos múltiples) como un gráfico acíclico dirigido  $G = (V; E)$ , llamado cálculo dag. Como ejemplo, la Figura 27.2 muestra el cálculo dag que resulta de calcular P-FIB.4/. Conceptualmente, los vértices en  $V$  son instrucciones y las aristas en  $E$  representan dependencias entre instrucciones, donde  $u; v \in V$  significa que la instrucción  $u$  debe ejecutarse antes que la instrucción  $v$ . Sin embargo, por conveniencia, si una cadena de instrucciones no contiene un control paralelo (sin generar, sincronizar o regresar de un generar, ya sea a través de una declaración de retorno explícita o el retorno que ocurre implícitamente al llegar al final de un procedimiento), podemos agrupar en una sola hebra, cada una de las cuales representa una o más instrucciones. Las instrucciones que involucran el control paralelo no se incluyen en los hilos, pero se representan en la estructura del dag. Por ejemplo, si una hebra tiene dos sucesores, uno de ellos debe haber sido generado, y una hebra con múltiples predecesores indica que los predecesores se unieron debido a una declaración de sincronización .

Así, en el caso general, el conjunto  $V$  forma el conjunto de hilos, y el conjunto  $E$  de aristas dirigidas representa las dependencias entre hilos inducidas por el control paralelo.

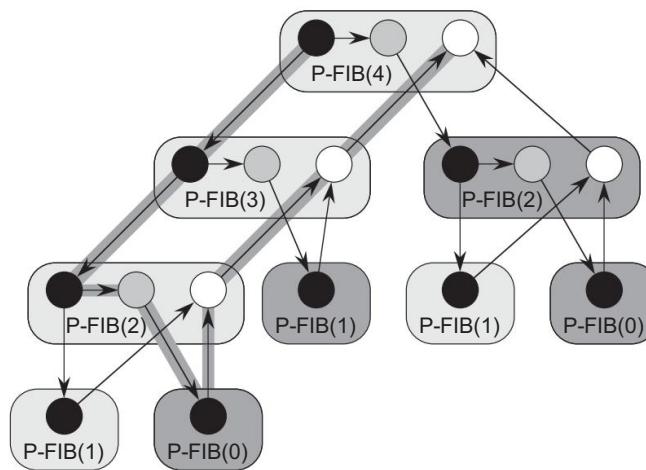


Figura 27.2 Gráfico acíclico dirigido que representa el cálculo de P-FIB<sub>4</sub>. Cada círculo representa una hebra, con círculos negros que representan casos base o la parte del procedimiento (instancia) hasta la generación de P-FIB<sub>n</sub> 1/ en la línea 3, círculos sombreados que representan la parte del procedimiento que llama P-FIB<sub>n</sub> 2/ en la línea 4 hasta la sincronización en la línea 5, donde se suspende hasta que vuelve la generación de P-FIB<sub>n</sub> 1/, y los círculos blancos representan la parte del procedimiento posterior a la sincronización donde suma x y y hasta el punto donde devuelve el resultado. Cada grupo de hebras que pertenecen al mismo procedimiento está rodeado por un rectángulo redondeado, ligeramente sombreado para los procedimientos generados y muy sombreado para los procedimientos llamados. Los bordes de generación y los bordes de llamada apuntan hacia abajo, los bordes de continuación apuntan horizontalmente hacia la derecha y los bordes de retorno apuntan hacia arriba. Suponiendo que cada cadena toma una unidad de tiempo, el trabajo es igual a 17 unidades de tiempo, ya que hay 17 cadenas, y el lapso es de 8 unidades de tiempo, ya que la ruta crítica, que se muestra con los bordes sombreados, contiene 8 cadenas.

Si G tiene un camino directo desde la hebra u hasta la hebra, decimos que las dos hebras están (lógicamente) en serie. De lo contrario, los hilos u y están (lógicamente) en paralelo.

Podemos imaginar un cómputo de subprocessos múltiples como un grupo de hebras incrustadas en un árbol de instancias de procedimientos. Por ejemplo, la Figura 27.1 muestra el árbol de instancias de procedimiento para P-FIB<sub>6</sub>/ sin la estructura detallada que muestra los hilos. La figura 27.2 amplía una sección de ese árbol y muestra las hebras que constituyen cada procedimiento. Todas las aristas dirigidas que conectan hebras se ejecutan dentro de un procedimiento oa lo largo de aristas no dirigidas en el árbol de procedimientos.

Podemos clasificar los bordes de un dag de computación para indicar el tipo de dependencias entre las diversas hebras. Una arista de continuación .u; u0 /, dibujada horizontalmente en la figura 27.2, conecta una hebra u con su sucesora u0 dentro del mismo procedimiento: el dag contiene un una hebra u genera una hebra que apunta hacia , borde de generación .u; /, instancia. Cuando abajo en la figura. Los bordes de llamada, que representan llamadas a procedimientos normales, también apuntan hacia abajo. La hebra de desove u difiere de la llamada u en que un desove induce un borde de continuación horizontal desde u hasta la hebra u0 siguiente.

mostrando  $u$  en su procedimiento, lo que indica que  $u_0$  es libre de ejecutarse al mismo tiempo, como , mientras que una llamada no induce tal borde. Cuando una hebra  $u$  vuelve a su procedimiento de llamada y  $x$  es la hebra que sigue inmediatamente a la siguiente sincronización en el procedimiento de llamada, el cálculo dag contiene el borde de retorno  $.u; x/$ , que apunta hacia arriba. Un cálculo comienza con una única hebra inicial, el vértice negro en el procedimiento denominado P-FIB.4/ en la figura 27.2, y termina con una única hebra final, el vértice blanco en el procedimiento denominado P-FIB.4/.

Estudiaremos la ejecución de algoritmos multiproceso en una computadora paralela ideal, que consiste en un conjunto de procesadores y una memoria compartida secuencialmente consistente . La coherencia secuencial significa que la memoria compartida, que en realidad puede estar realizando muchas cargas y almacenamientos desde los procesadores al mismo tiempo, produce los mismos resultados que si en cada paso se ejecutara exactamente una instrucción de uno de los procesadores. Es decir, la memoria se comporta como si las instrucciones se ejecutaran secuencialmente según algún orden lineal global que conserva los órdenes individuales en los que cada procesador emite sus propias instrucciones. Para los cálculos dinámicos de subprocesos múltiples, que la plataforma de concurrencia programa automáticamente en los procesadores, la memoria compartida se comporta como si las instrucciones del cálculo de subprocesos múltiples estuvieran intercaladas para producir un orden lineal que conserva el orden parcial del dag de cálculo. Dependiendo de la programación, el orden puede diferir de una ejecución del programa a otra, pero el comportamiento de cualquier ejecución puede entenderse asumiendo que las instrucciones se ejecutan en algún orden lineal consistente con el cálculo dag.

Además de hacer suposiciones sobre la semántica, el modelo de computadora ideal en paralelo hace algunas suposiciones de rendimiento. Específicamente, asume que cada procesador en la máquina tiene la misma potencia de cómputo e ignora el costo de la programación. Aunque esta última suposición puede parecer optimista, resulta que para algoritmos con suficiente "paralelismo" (un término que definiremos con precisión en un momento), la sobrecarga de programación es generalmente mínima en la práctica.

### Medidas de desempeño

Podemos medir la eficiencia teórica de un algoritmo de subprocesos múltiples mediante el uso de dos métricas: "trabajo" y "intervalo". El trabajo de un cómputo multiproceso es el tiempo total para ejecutar todo el cómputo en un procesador. En otras palabras, el trabajo es la suma de los tiempos que toma cada uno de los hilos. Para un dag de computación en el que cada hebra toma una unidad de tiempo, el trabajo es solo el número de vértices en el dag. El lapso es el tiempo más largo para ejecutar los hilos a lo largo de cualquier camino en el dag. Nuevamente, para un dag en el que cada hebra toma una unidad de tiempo, el lapso es igual al número de vértices en un camino más largo o crítico en el dag. (Recuerde de la Sección 24.2 que podemos encontrar una ruta crítica en un dag GD .V; E/ en ,VCE/ tiempo.) Por ejemplo, el cálculo dag de la Figura 27.2 tiene 17 vértices en total y 8 vértices en su punto crítico.

camino, de modo que si cada hebra toma una unidad de tiempo, su trabajo es de 17 unidades de tiempo y su lapso es de 8 unidades de tiempo.

El tiempo de ejecución real de un cómputo de subprocessos múltiples depende no solo de su trabajo y su duración, sino también de cuántos procesadores están disponibles y cómo el planificador asigna los hilos a los procesadores. Para denotar el tiempo de ejecución de un cálculo multihilo en  $P$  procesadores, subíndicemos con  $P$ . Por ejemplo, podríamos denotar el tiempo de ejecución de un algoritmo en  $P$  procesadores con  $TP$ . El trabajo es el tiempo de ejecución en un solo procesador, o  $T_1$ . El lapso es el tiempo de ejecución si pudiéramos ejecutar cada hebra en su propio procesador, en otras palabras, si tuviéramos un número ilimitado de procesadores, por lo que denotamos el lapso por  $T_1$ .

El trabajo y el lapso proporcionan límites inferiores en el tiempo de ejecución  $TP$  de un multi Cálculo de subprocessos en procesadores  $P$ :

En un paso, una computadora paralela ideal con procesadores  $P$  puede realizar como máximo  $P$  unidades de trabajo y, por lo tanto, en tiempo  $TP$ , puede realizar como máximo trabajo  $P \cdot TP$ .

Como el trabajo total a realizar es  $T_1$ , tenemos  $P \cdot TP \geq T_1$ . Dividiendo por  $P$  se obtiene la ley del trabajo:

$$TP \leq T_1 / P \quad (27.2)$$

Una computadora paralela ideal con procesador  $P$  no puede funcionar más rápido que una máquina con un número ilimitado de procesadores. Visto de otra manera, una máquina con un número ilimitado de procesadores puede emular una máquina con procesador  $P$  usando solo  $P$  de sus procesadores. Por lo tanto, la ley de amplitud sigue:

$$TP \geq T_1 / P \quad (27.3)$$

Definimos la aceleración de un cálculo en  $P$  procesadores por la relación  $A = T_1 / TP$ , que indica cuántas veces más rápido es el cálculo en  $P$  procesadores que en 1 procesador. Por la ley del trabajo, tenemos  $TP \leq T_1 / P$ , lo que implica que  $A \geq 1/P$ . Así, la aceleración en  $P$  procesadores puede ser como máximo  $1/P$ . Cuando la aceleración es lineal en el número de procesadores, es decir, cuando  $A = (T_1 / TP) \cdot P$ , el cálculo exhibe una aceleración lineal y cuando  $A = T_1 / TP$ , tenemos una aceleración lineal perfecta.

La relación  $A = T_1 / TP$  da el paralelismo del cálculo de subprocessos múltiples. Podemos ver el paralelismo desde tres perspectivas. Como proporción, el paralelismo denota la cantidad promedio de trabajo que se puede realizar en paralelo para cada paso a lo largo de la ruta crítica. Como límite superior, el paralelismo proporciona la máxima aceleración posible que se puede lograr en cualquier número de procesadores. Finalmente, y quizás lo más importante, el paralelismo proporciona un límite a la posibilidad de lograr una aceleración lineal perfecta. Específicamente, una vez que el número de procesadores excede el paralelismo, la computación no puede lograr una aceleración lineal perfecta. Para ver este último punto, supongamos que  $P > T_1 / TP$ , en cuyo caso

la ley de expansión implica que la aceleración satisface  $T_1=TP$   $T_1=T_1 < P$ . Además, si el número  $P$  de procesadores en la computadora paralela ideal excede en gran medida el paralelismo, es decir, si  $P T_1=T_1$ , entonces  $T_1=TP P$ , por lo que la aceleración es mucho menor que la cantidad de procesadores. En otras palabras, cuantos más procesadores usemos más allá del paralelismo, menos perfecta será la aceleración.

Como ejemplo, considere el cálculo P-FIB.4/ en la figura 27.2 y suponga que cada hebra toma una unidad de tiempo. Como el trabajo es  $T_1 D 17$  y el tramo es  $T_1 D 8$ , el paralelismo es  $T_1=T_1 D 17=8 D 2:125$ . En consecuencia, lograr mucho más del doble de la aceleración es imposible, sin importar cuántos procesadores empleemos para ejecutar el cálculo. Sin embargo, para tamaños de entrada más grandes, veremos que P-FIB.n/ exhibe un paralelismo sustancial.

Definimos la holgura (paralela) de un cálculo de subprocesos múltiples ejecutado en una computadora paralela ideal con  $P$  procesadores como la relación  $.T_1=T_1/P D T_1=.P T_1/$ , que es el factor por el cual el paralelismo del cálculo excede el número de procesadores en la máquina. Por lo tanto, si la holgura es menor que 1, no podemos esperar lograr una aceleración lineal perfecta, porque  $T_1=.P T_1/ < 1$  y la ley de expansión implican que la aceleración en los procesadores  $P$  satisface  $T_1=TP$   $T_1=T_1 < P$ .

De hecho, a medida que la holgura disminuye de 1 a 0, la aceleración del cálculo diverge cada vez más de la aceleración lineal perfecta. Sin embargo, si la holgura es mayor que 1, el trabajo por procesador es la restricción limitante. Como veremos, a medida que la holgura aumenta desde 1, un buen programador puede lograr una aceleración lineal cada vez más cercana a la perfección.

### Planificación

El buen desempeño depende de algo más que de minimizar el trabajo y la duración. Los hilos también deben programarse eficientemente en los procesadores de la máquina paralela.

Nuestro modelo de programación de subprocesos múltiples no proporciona ninguna forma de especificar qué hebras ejecutar en qué procesadores. En su lugar, confiamos en el programador de la plataforma de concurrencia para mapear el cómputo que se desarrolla dinámicamente a los procesadores individuales. En la práctica, el programador asigna los hilos a hilos estáticos y el sistema operativo programa los hilos en los propios procesadores, pero este nivel adicional de direccionamiento indirecto es innecesario para nuestra comprensión de la programación. Podemos imaginarnos que el programador de la plataforma de concurrencia asigna hebras a procesadores directamente.

Un programador multiproceso debe programar el cálculo sin conocimiento previo de cuándo se generarán los hilos o cuándo se completarán; debe operar en línea. Además, un buen programador opera de forma distribuida, donde los subprocesos que implementan el programador cooperan para equilibrar la carga del cálculo. Existen planificadores distribuidos en línea demostrablemente buenos, pero analizarlos es complicado.

En su lugar, para simplificar nuestro análisis, investigaremos un planificador centralizado en línea, que conoce el estado global de la computación en un momento dado. En particular, analizaremos los programadores codiciosos, que asignan tantos hilos a los procesadores como sea posible en cada paso de tiempo. Si al menos  $P$  hebras están listas para ejecutarse durante un paso de tiempo, decimos que el paso es un paso completo, y un planificador codicioso asigna cualquier  $P$  de las hebras listas a los procesadores. De lo contrario, menos de los hilos  $P$  están listos para ejecutarse, en cuyo caso decimos que el paso es un paso incompleto y el programador asigna cada hilo listo a su propio procesador.

Según la ley de trabajo, el mejor tiempo de ejecución que podemos esperar en los procesadores  $P$  es  $TP \leq D \leq T_1 = P$ , y según la ley de amplitud, lo mejor que podemos esperar es  $TP \geq D \geq T_1$ . El siguiente teorema muestra que la programación codiciosa es demostrablemente buena porque logra la suma de estos dos límites inferiores como un límite superior.

#### Teorema 27.1

En una computadora paralela ideal con procesadores  $P$ , un programador codicioso ejecuta un cálculo de subprocessos múltiples con trabajo  $T_1$  y intervalo  $T_1$  en el tiempo

$$TP \leq D \leq T_1 = P \quad (27.4)$$

Prueba Comenzamos considerando los pasos completos. En cada paso completo, los procesadores  $P$  juntos realizan un trabajo  $P$  total. Supongamos, a efectos de contradicción, que el número de pasos completos es estrictamente mayor que  $bT_1 = P$ .

Entonces, el trabajo total de los pasos completos es al menos

$$P \cdot bT_1 = P \geq C / DP \geq bT_1 = P \geq CPD \geq T_1 \cdot T_1 \text{ mod}$$

$$P / CP \text{ (por ecuación (3.8))}$$

$$> T_1 \text{ (por la desigualdad (3.9))} .$$

Así, obtenemos la contradicción de que los procesadores  $P$  realizarían más trabajo del que requiere el cómputo, lo que nos permite concluir que el número de pasos completos es como máximo  $bT_1 = P$ .

Ahora, considere un paso incompleto. Sea  $G$  el dag que representa el cómputo completo y, sin pérdida de generalidad, suponga que cada hebra toma una unidad de tiempo. (Podemos reemplazar cada hebra más larga por una cadena de hebras de unidad de tiempo.) Sea  $G_0$  el subgrafo de  $G$  que aún debe ejecutarse al comienzo del paso incompleto, y sea  $G_{00}$  el subgrafo que queda por ejecutar después del paso incompleto. Una ruta más larga en un dag debe necesariamente comenzar en un vértice con grado de entrada 0. Dado que un paso incompleto de un planificador codicioso ejecuta todos los hilos con grado de entrada 0 en  $G_0$ , la longitud de la ruta más larga en  $G_{00}$  debe ser 1 menos que el longitud de un camino más largo en  $G_0$ . En otras palabras, un paso incompleto reduce el lapso del dag no ejecutado en 1. Por lo tanto, el número de pasos incompletos es como máximo  $T_1$ .

Dado que cada paso es completo o incompleto, se sigue el teorema. ■

El siguiente corolario del Teorema 27.1 muestra que un programador codicioso siempre se desempeña bien.

**Corolario 27.2 El**

tiempo de ejecución TP de cualquier cálculo multiproceso programado por un planificador codicioso en una computadora paralela ideal con procesadores P está dentro de un factor de 2 del óptimo.

Prueba Sea T con<sub>max</sub> sea el tiempo de ejecución producido por un programador óptimo en una máquina P procesadores, y sean T1 y T1 el trabajo y el lapso del cálculo, respectivamente. Dado que las leyes del trabajo y de la longitud —desigualdades (27.2) y (27.3)— dan max.T1=P; T1/, el teorema 27.1 implica que us TP

$$\begin{aligned} \text{TP} &= \text{PC } T1/2 \\ &\leq \max.T1=P; T1/2T \end{aligned}$$

pag :

■

El siguiente corolario muestra que, de hecho, un programador codicioso logra resultados casi perfectos. aceleración lineal en cualquier cálculo de subprocesos múltiples a medida que crece la holgura.

**Corolario 27.3 Sea**

TP el tiempo de ejecución de un cómputo de subprocesos múltiples producido por un planificador codicioso en una computadora paralela ideal con procesadores P, y sean T1 y T1 el trabajo y la duración del cómputo, respectivamente. Entonces, si PT1 =T1, tenemos TP T1=P, o de manera equivalente, una aceleración de aproximadamente P.

Prueba Si suponemos que P T1=T1, entonces también tenemos T1 T1=P, y T1=PC T1 T1=P. Dado que el concluimos que TP T1=P, o la ley equivalente trabajo, por lo tanto, el Teorema 27.1 nos da TP T1=P, (27.2) dicta que TP lento, que la aceleración es T1=TP P.

■

El símbolo denota "mucho menos", pero ¿cuánto es "mucho menos"? Como regla general, una holgura de al menos 10, es decir, 10 veces más paralelismo que los procesadores, generalmente es suficiente para lograr una buena aceleración. Entonces, el término de amplitud en el límite codicioso, la desigualdad (27.4), es menos del 10 % del término de trabajo por procesador, que es lo suficientemente bueno para la mayoría de las situaciones de ingeniería. Por ejemplo, si un cómputo se ejecuta en solo 10 o 100 procesadores, no tiene sentido valorar el paralelismo de, digamos, 1,000,000 sobre el paralelismo de 10,000, incluso con el factor de 100 de diferencia. Como muestra el problema 27-2, a veces al reducir el paralelismo extremo, podemos obtener algoritmos que son mejores con respecto a otras preocupaciones y que aún se pueden escalar bien en un número razonable de procesadores.

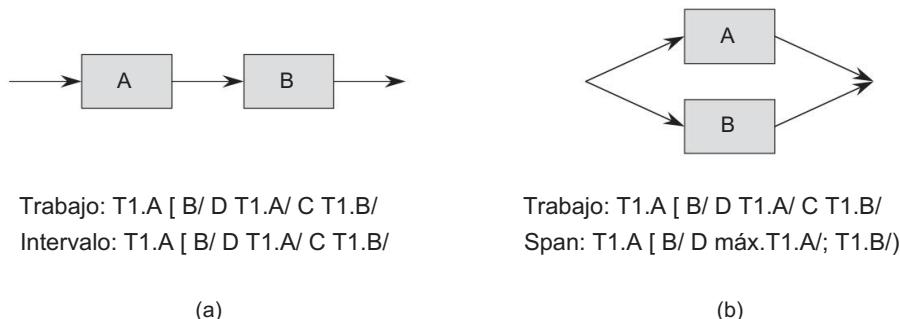


Figura 27.3 El trabajo y la duración de los subcálculos compuestos. (a) Cuando dos subcálculos se unen en serie, el trabajo de la composición es la suma de sus trabajos, y el lapso de la composición es la suma de sus lapsos. (b) Cuando dos subcálculos se unen en paralelo, el trabajo de la composición sigue siendo la suma de su trabajo, pero el lapso de la composición es solo el máximo de sus lapsos.

## Análisis de algoritmos multiproceso

Ahora tenemos todas las herramientas que necesitamos para analizar algoritmos de subprocesos múltiples y proporcionar buenos límites en sus tiempos de ejecución en varios procesadores. Analizar el trabajo es relativamente sencillo, ya que no es más que analizar el tiempo de ejecución de un algoritmo en serie ordinario, es decir, la serialización del algoritmo de subprocesos múltiples, con el que ya debería estar familiarizado, ya que eso es lo que trata la mayor parte de este libro de texto. ¡es sobre! Analizar el lapso es más interesante, pero generalmente no es más difícil una vez que aprendes a hacerlo.

Investigaremos las ideas básicas usando el programa P-FIB.

Analizar el trabajo T1.n/ de P-FIB.n/ no supone ningún obstáculo, porque ya lo hemos hecho. El procedimiento FIB original es esencialmente la serialización de P-FIB y, por lo tanto, T1.n/ DT.n/D.n/ de la ecuación (27.1)

La figura 27.3 ilustra cómo analizar el lapso. Si dos subcálculos se unen en serie, sus tramos se suman para formar el tramo de su composición, mientras que si se unen en paralelo, el tramo de su composición es el máximo de los tramos de los dos subcálculos. Para P-FIB.n/, la llamada generada a P-FIB.n1/ en la línea 3 se ejecuta en paralelo con la llamada a P-FIB.n 2/ en la línea 4. Por lo tanto, podemos expresar el intervalo de P-FIB .n/ como la recurrencia

T1.n/ D máx.T1.n 1/; T1.n 2// C ,.1/  
D T1.n 1/ C ..1/ :

que tiene solución  $T_1 \cdot n / P \cdot n /$ .

El paralelismo de P-FIB.n es  $T1.n = T1.n / D$ ,  $n=n!$ , que crece dramáticamente a medida que n crece. Por lo tanto, incluso en las computadoras paralelas más grandes, una modesta

El valor de n es suficiente para lograr una aceleración lineal casi perfecta para P-FIB.n/, porque este procedimiento exhibe una holgura paralela considerable.

### Bucles paralelos

Muchos algoritmos contienen bucles cuyas iteraciones pueden funcionar en paralelo. Como veremos, podemos paralelizar dichos bucles usando las palabras clave spawn y sync , pero es mucho más conveniente especificar directamente que las iteraciones de tales bucles pueden ejecutarse simultáneamente. Nuestro pseudocódigo proporciona esta funcionalidad a través de la palabra clave de concurrencia paralela , que precede a la palabra clave for en una instrucción de bucle for .

Como ejemplo, considere el problema de multiplicar una matriz nn AD .aij / por un n-vector x D .xj /. El n-vector resultante y D .yi/ viene dado por la ecuación

$$yi \leftarrow \sum_{j=1}^n a_{ij} x_j;$$

para i D 1; 2; : : : ; norte. Podemos realizar la multiplicación matriz-vector calculando todas las entradas de y en paralelo de la siguiente manera:

```

MAT-VEC.A; x/ 1 n
D A: filas 2 sea y
un nuevo vector de longitud n 3 paralelo
para i D 1 a n 4 yi D 0 5 paralelo
para i D 1 a n 6
para j D 1 a n yi D yi C aij xj 8
volver y
7

```

En este código, las palabras clave paralelas for en las líneas 3 y 5 indican que las iteraciones de los bucles respectivos pueden ejecutarse simultáneamente. Un compilador puede implementar cada bucle for paralelo como una subrutina de dividir y vencer usando el paralelismo anidado. Por ejemplo, el bucle for paralelo en las líneas 5 a 7 se puede implementar con la llamada MAT-VEC-MAIN-LOOP.A; X; y; norte; 1; n/, donde el compilador produce la subrutina auxiliar MAT-VEC-MAIN-LOOP de la siguiente manera:

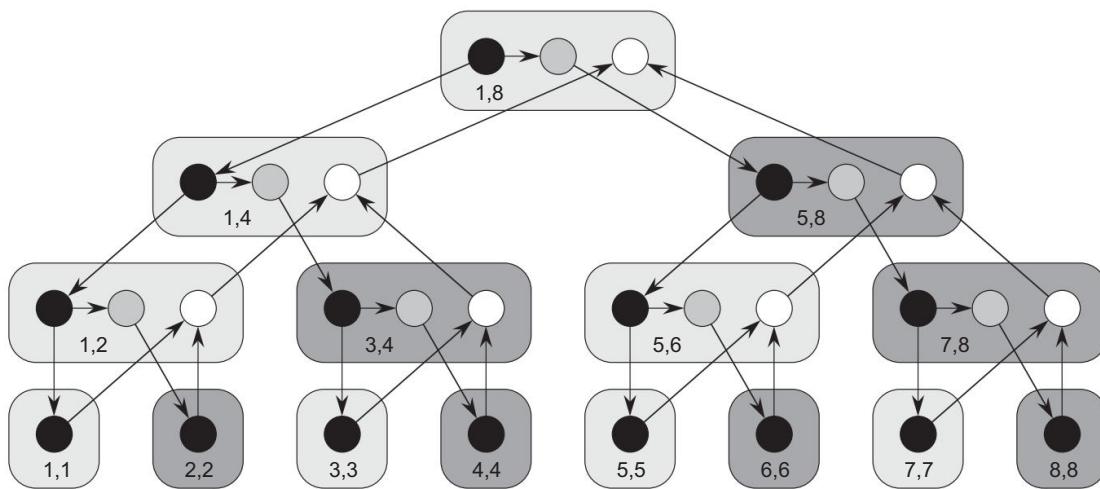


Figura 27.4 Un dag que representa el cálculo de MAT-VEC-MAIN-LOOP.A; X; y; 8; 1; 8/. Los dos números dentro de cada rectángulo redondeado dan los valores de los dos últimos parámetros (i e i0 en el encabezado del procedimiento) en la invocación (generación o llamada) del procedimiento. Los círculos negros representan hebras correspondientes al caso base oa la parte del procedimiento hasta la generación de MAT-VEC-MAIN-LOOP en la línea 5; los círculos sombreados representan hebras correspondientes a la parte del procedimiento que llama a MAT-VEC-MAIN-LOOP en la línea 6 hasta la sincronización en la línea 7, donde se suspende hasta que regresa la subrutina generada en la línea 5; y los círculos blancos representan hebras correspondientes a la parte (insignificante) del procedimiento después de la sincronización hasta el punto donde regresa.

```

MAT-VEC-MAIN-LOOP.A; X; y; norte; i; i0 / 1
si i == i0 para
2      j D 1 a n
3 yi D yi C aij xj 4 else mid D bi C
i0 /=2c 5 spawn MAT-VEC-MAIN-
LOOP.A; X; y; norte; i; mid/ 6 MAT-VEC-MAIN-LOOP.A; X; y; norte;
do medio 1; i0 /
7      sincronizar

```

Este código genera recursivamente la primera mitad de las iteraciones del ciclo para ejecutar en paralelo con la segunda mitad de las iteraciones y luego ejecuta una sincronización, creando así un árbol binario de ejecución donde las hojas son iteraciones de ciclo individuales, como se muestra en la Figura 27.4 .

Para calcular el trabajo T1.n/ de MAT-VEC en una matriz nn, simplemente calculamos el tiempo de ejecución de su serialización, que obtenemos reemplazando los bucles for paralelos por bucles for ordinarios . Por lo tanto, tenemos T1.n/ D ,n2/, porque domina el tiempo de ejecución cuadrático de los bucles doblemente anidados en las líneas 5–7. Este análisis

Sin embargo, parece ignorar la sobrecarga del desove recursivo al implementar los bucles paralelos. De hecho, la sobrecarga del desove recursivo aumenta el trabajo de un ciclo paralelo en comparación con el de su serialización, pero no asintóticamente.

Para ver por qué, observe que dado que el árbol de instancias de procedimientos recursivos es un árbol binario completo, el número de nodos internos es 1 menos que el número de hojas (vea el Ejercicio B.5-3). Cada nodo interno realiza un trabajo constante para dividir el rango de iteración, y cada hoja corresponde a una iteración del ciclo, que toma al menos un tiempo constante ( $.n$  tiempo en este caso). Por lo tanto, podemos amortizar la sobrecarga del desove recursivo contra el trabajo de las iteraciones, contribuyendo como máximo con un factor constante al trabajo general.

En la práctica, las plataformas de concurrencia de subprocesos múltiples dinámicos a veces engrosan las hojas de la recursividad ejecutando varias iteraciones en una sola hoja, ya sea automáticamente o bajo el control del programador, lo que reduce la sobrecarga del desove recursivo. Sin embargo, esta sobrecarga reducida se produce a expensas de reducir también el paralelismo, pero si el cálculo tiene suficiente holgura paralela, no es necesario sacrificar una aceleración lineal casi perfecta.

También debemos tener en cuenta la sobrecarga del desove recursivo al analizar el lapso de una construcción de bucle paralelo. Dado que la profundidad de la llamada recursiva es logarítmica en el número de iteraciones, para un bucle paralelo con  $n$  iteraciones en las que la  $i$ -ésima iteración tiene un intervalo  $\text{iter1}.i /$ , el intervalo es

$$T1.n / D , \lg n / C \text{ máx} \quad \text{iter1}.i : \\ 1 \text{ en}$$

Por ejemplo, para MAT-VEC en una matriz  $nn$ , el bucle de inicialización paralelo en las líneas 3 y 4 tiene un intervalo  $, \lg n /$ , porque la generación recursiva domina el trabajo de tiempo constante de cada iteración. El lapso de los bucles doblemente anidados en las líneas 5 a 7 es  $,n$ , porque cada iteración del bucle for paralelo exterior contiene  $n$  iteraciones del bucle for interior (serie). El lapso del código restante en el procedimiento es constante y, por lo tanto, el lapso está dominado por los bucles doblemente anidados, lo que produce un lapso general de  $,n$  para todo el procedimiento. Como el trabajo es  $,n^2 /$ , el paralelismo es  $,n^2 = ,n / D ,n /$ . (El ejercicio 27.1-6 le pide que proporcione una implementación con aún más paralelismo).

#### Condiciones de carrera

Un algoritmo de subprocesos múltiples es determinista si siempre hace lo mismo en la misma entrada, sin importar cómo estén programadas las instrucciones en la computadora multinúcleo. No es determinista si su comportamiento puede variar de una ejecución a otra. A menudo, un algoritmo de subprocesos múltiples que pretende ser determinista no lo es porque contiene una "carrera de determinación".

Las condiciones de carrera son la pesadilla de la concurrencia. Los errores de carrera famosos incluyen la máquina de radioterapia Therac-25, que mató a tres personas e hirió a varias.

otros, y el apagón norteamericano de 2003, que dejó a más de 50 millones de personas sin electricidad. Estos errores perniciosos son notoriamente difíciles de encontrar. Puede ejecutar pruebas en el laboratorio durante días sin fallar solo para descubrir que su software falla esporádicamente en el campo.

Una carrera de determinación ocurre cuando dos instrucciones lógicamente paralelas acceden a la misma ubicación de memoria y al menos una de las instrucciones realiza una escritura. El siguiente procedimiento ilustra una condición de carrera:

```
CARRERA-EJEMPLO. /
1 x D 0 2
paralelo para i D 1 a 2 3 x D x
C 1 4 imprimir x
```

Después de inicializar  $x$  a 0 en la línea 1, RACE-EXAMPLE crea dos hilos paralelos, cada uno de los cuales incrementa  $x$  en la línea 3. Aunque podría parecer que RACE EXAMPLE siempre debería imprimir el valor 2 (su serialización ciertamente lo hace), en su lugar podría imprimir el valor 1. Veamos cómo puede ocurrir esta anomalía.

Cuando un procesador incrementa  $x$ , la operación no es indivisible, sino que se compone de una secuencia de instrucciones:

1. Lea  $x$  de la memoria en uno de los registros del procesador.
2. Incrementar el valor en el registro.
3. Vuelva a escribir el valor del registro en  $x$  en la memoria.

La figura 27.5(a) ilustra un dag de cálculo que representa la ejecución de RACE EXAMPLE, con los hilos desglosados en instrucciones individuales. Recuerde que dado que una computadora paralela ideal admite la consistencia secuencial, podemos ver la ejecución paralela de un algoritmo de subprocessos múltiples como un entrelazado de instrucciones que respeta las dependencias en el dag. La parte (b) de la figura muestra los valores en una ejecución del cálculo que provoca la anomalía. El valor  $x$  se almacena en la memoria y  $r1$  y  $r2$  son registros del procesador. En el paso 1, uno de los procesadores establece  $x$  en 0. En los pasos 2 y 3, el procesador 1 lee  $x$  de la memoria en su registro  $r1$  y lo incrementa, produciendo el valor 1 en  $r1$ . En ese momento, el procesador 2 entra en escena y ejecuta las instrucciones 4–6. El procesador 2 lee  $x$  de la memoria en el registro  $r2$ ; lo incrementa, produciendo el valor 1 en  $r2$ ; y luego almacena este valor en  $x$ , configurando  $x$  en 1. Ahora, el procesador 1 continúa con el paso 7, almacenando el valor 1 en  $r1$  en  $x$ , lo que deja el valor de  $x$  sin cambios. Por lo tanto, el paso 8 imprime el valor 1, en lugar de 2, como lo haría la serialización.

Podemos ver lo que ha sucedido. Si el efecto de la ejecución en paralelo fuera que el procesador 1 ejecutara todas sus instrucciones antes que el procesador 2, el valor 2 sería

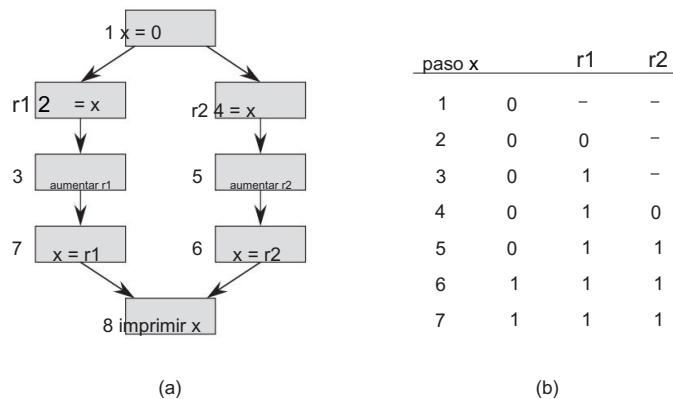


Figura 27.5 Ilustración de la carrera de determinación en RACE-EXAMPLE. (a) Un dag de cálculo que muestra las dependencias entre instrucciones individuales. Los registros del procesador son  $r1$  y  $r2$ . Se omiten las instrucciones no relacionadas con la carrera, como la implementación del control de bucle. (b) Una secuencia de ejecución que provoca el error, mostrando los valores de  $x$  en la memoria y registra  $r1$  y  $r2$  para cada paso en la secuencia de ejecución.

impreso. Por el contrario, si el efecto fuera que el procesador 2 ejecutara todas sus instrucciones antes que el procesador 1, el valor 2 aún se imprimiría. Sin embargo, cuando las instrucciones de los dos procesadores se ejecutan al mismo tiempo, es posible, como en la ejecución de este ejemplo, que se pierda una de las actualizaciones de x.

Por supuesto, muchas ejecuciones no provocan el error. Por ejemplo, si la orden de ejecución fuera h1; 2; 3; 7; 4; 5; 6; 8i o h1; 4; 5; 6; 2; 3; 7; 8i, obtendríamos el resultado correcto. Ese es el problema con las carreras de determinación. Generalmente, la mayoría de los pedidos producen resultados correctos, como aquellos en los que las instrucciones de la izquierda se ejecutan antes que las instrucciones de la derecha, o viceversa. Pero algunos pedidos generan resultados inadecuados cuando las instrucciones se intercalan. En consecuencia, las carreras pueden ser extremadamente difíciles de probar. Puede ejecutar pruebas durante días y nunca ver el error, solo para experimentar un colapso catastrófico del sistema en el campo cuando el resultado es crítico.

Aunque podemos hacer frente a las carreras en una variedad de formas, incluido el uso de bloqueos de exclusión mutua y otros métodos de sincronización, para nuestros propósitos, simplemente nos aseguraremos de que los hilos que operan en paralelo sean independientes: no tienen carreras de determinación entre ellos. Por lo tanto, en un paralelo para construir, todas las iteraciones deben ser independientes. Entre un engendro y la sincronización correspondiente, el código del hijo engendrado debe ser independiente del código del padre, incluido el código ejecutado por hijos engendrados o llamados adicionales. Tenga en cuenta que los argumentos para un hijo generado se evalúan en el padre antes de que ocurra la generación real y, por lo tanto, la evaluación de los argumentos para una subrutina generada está en serie con cualquier acceso a esos argumentos después de la generación.

Como ejemplo de lo fácil que es generar código con carreras, aquí hay una implementación defectuosa de la multiplicación de vector de matriz de subprocessos múltiples que logra un lapso de  $\lg n$  al paralelizar el bucle for interno :

```
MAT-VEC-INCORRECTO.A; x/ 1
n D A: filas 2 sea y
un nuevo vector de longitud n 3 paralelo para
i D 1 a n 4 yi D 0 5 paralelo para i
D 1 a n 6 paralelo
para j D 1 a n 7 yi D yi C aij xj 8
retorno y
```

Desafortunadamente, este procedimiento es incorrecto debido a las carreras en la actualización de  $y_i$  en la línea 7, que se ejecuta simultáneamente para todos los  $n$  valores de  $j$ . El ejercicio 27.1-6 le pide que proporcione una implementación correcta con  $\lg n / \text{span}$ .

Un algoritmo de subprocessos múltiples con carreras a veces puede ser correcto. Como ejemplo, dos subprocessos paralelos pueden almacenar el mismo valor en una variable compartida, y no importaría cuál almacenó el valor primero. En general, sin embargo, consideraremos que el código con razas es ilegal.

una lección de ajedrez

Cerramos esta sección con una historia real que ocurrió durante el desarrollo del programa de juego de ajedrez de subprocessos múltiples de clase mundial ?Socrates [80], aunque los tiempos a continuación se han simplificado para la exposición. El programa se prototípó en una computadora de 32 procesadores, pero finalmente se ejecutaría en una supercomputadora con 512 procesadores. En un momento, los desarrolladores incorporaron una optimización en el programa que redujo su tiempo de ejecución en un punto de referencia importante en la máquina de 32 procesadores de T32 D 65 segundos a TD 40 segundos. Sin embargo, los desarrolladores utilizaron las medidas de rendimiento del trabajo y del intervalo para concluir que la versión optimizada, que era más rápida en 32 procesadores, en realidad sería más lenta que la versión original en 512 procesadores. Como resultado, abandonaron la "optimización".

Aquí está su análisis. La versión original del programa tenía trabajo  $T_1 D 2048$  segundos y lapso  $T_1 D 1$  segundo. Si tratamos la desigualdad (27.4) como una ecuación,  $TP D T_1=PC T_1$ , y la usamos como una aproximación al tiempo de ejecución en los procesadores  $P$ , vemos que, de hecho,  $T_{32} D 2048=32 C 1 D 65$ . Con la optimización , el D 8 segundos. De nuevo el trabajo se convirtió en  $T_1^0 D 1024$  segundos y el lapso se convirtió en  $TD_1^0$  nuestra aproximación, obtenemos  $T_1 \sin \frac{0}{32} 1024=32 C 8 D 40$ . Usando embargo, las velocidades relativas de las dos versiones cambian cuando calculamos los tiempos de ejecución en 512 procesadores. En particular tenemos  $T_{512} D 2048=512C_1 D 5$

segundos, y  $TD = \frac{C}{P} = \frac{512}{8} = 64$  segundos. ¡La optimización que aceleró el programa en 32 procesadores habría hecho que el programa fuera el doble de lento en 512 procesadores! El lapso de 8 de la versión optimizada, que no era el término dominante en el tiempo de ejecución en 32 procesadores, se convirtió en el término dominante en 512 procesadores, anulando la ventaja de usar más procesadores.

La moraleja de la historia es que el trabajo y la duración pueden proporcionar un mejor medio de extrapolando el rendimiento de lo que pueden medir los tiempos de funcionamiento.

### Ejercicios

#### 27.1-1

Supongamos que generamos P-FIB.n 2/ en la línea 4 de P-FIB, en lugar de llamarlo como se hace en el código. ¿Cuál es el impacto en el trabajo asintótico, el intervalo y el paralelismo?

#### 27.1-2

Dibujar el dag de cómputo que resulta de ejecutar P-FIB.5/. Suponiendo que cada hebra en el cómputo toma una unidad de tiempo, ¿cuáles son el trabajo, el lapso y el paralelismo del cómputo? Muestre cómo programar el dag en 3 procesadores utilizando la programación codiciosa etiquetando cada hebra con el paso de tiempo en el que se ejecuta.

#### 27.1-3

Demuestre que un programador codicioso logra el siguiente límite de tiempo, que es ligeramente más fuerte que el límite demostrado en el Teorema 27.1:

$$TP = \frac{T_1 T_1}{T_1 + T_1} C T_1 : \quad (27.5)$$

#### 27.1-4

Construya un dag de cómputo para el cual una ejecución de un programador voraz pueda tomar casi el doble de tiempo que otra ejecución de un programador voraz en la misma cantidad de procesadores. Describa cómo procederían las dos ejecuciones.

#### 27.1-5

La profesora Karan mide su algoritmo determinista de subprocesos múltiples en 4, 10 y 64 procesadores de una computadora paralela ideal usando un planificador codicioso. Ella afirma que las tres carreras produjeron T4 D 80 segundos, T10 D 42 segundos y T64 D 10 segundos. Argumentar que el profesor miente o es incompetente. (Sugerencia: utilice la ley del trabajo (27.2), la ley del intervalo (27.3) y la desigualdad (27.5) del ejercicio 27.1-3).

## 27.1-6

Proporcione un algoritmo de subprocessos múltiples para multiplicar una matriz  $n \times n$  por un vector  $n$  que logre un paralelismo  $\sqrt{n} = \lg n$  mientras mantiene el trabajo  $\sqrt{n}$ .

## 27.1-7

Considere el siguiente pseudocódigo de subprocessos múltiples para transponer una matriz  $A$   $n \times n$  en su lugar:

P-TRANSPOSER.A/ 1

$n$  D  $A$ : filas 2

paralelas para  $j$  D 2 a  $n$  3 paralelas

para  $i$  D 1 a  $j$  4 intercambiar  $a_{ij}$  con  $a_{ji}$

Analice el trabajo, el lapso y el paralelismo de este algoritmo.

## 27.1-8

Suponga que reemplazamos el bucle `for` paralelo en la línea 3 de P-TRANSPOSE (vea el ejercicio 27.1-7) con un bucle `for` ordinario. Analice el trabajo, el intervalo y el paralelismo del algoritmo resultante.

## 27.1-9

¿ Para cuántos procesadores las dos versiones de los programas de ajedrez se ejecutan a la misma velocidad, suponiendo que  $T_P = T_D = T_C$  ?

## 27.2 Multiplicación de matrices de subprocessos múltiples

En esta sección, examinamos cómo realizar múltiples subprocessos en la multiplicación de matrices, un problema cuyo tiempo de ejecución en serie estudiamos en la Sección 4.2. Veremos los algoritmos de subprocessos múltiples basados en el bucle anidado triple estándar, así como los algoritmos de divide y vencerás.

### Multiplicación de matrices multiproceso

El primer algoritmo que estudiamos es el algoritmo simple basado en paralelizar los bucles en el procedimiento SQUARE-MATRIX-MULTIPLY en la página 75:

**P-MATRIZ CUADRADA-MULTIPLICAR.A; B/**

1 n D A: filas 2 sea  
 C una nueva matriz nn 3 paralela  
 para i D 1 a n 4 paralela para j  
 D 1 a n 5 cij D 0 6 para k D 1 a n 7  
 cij D cij C aik bkj 8  
 vuelta C

Para analizar este algoritmo, observe que dado que la serialización del algoritmo es solo SQUARE-MATRIX-MULTIPLY, el trabajo es simplemente T1.n/ D ,n3/, lo mismo que el tiempo de ejecución de SQUARE-MATRIX-MULTIPLY. El lapso es T1.n/ D ,n/, porque sigue un camino hacia abajo en el árbol de recursividad para el ciclo for paralelo que comienza en la línea 3, luego hacia abajo en el árbol de recursión para el ciclo for paralelo que comienza en la línea 4, y luego ejecuta todas las iteraciones n del bucle for ordinario que comienza en la línea 6, lo que da como resultado un intervalo total de ..lg n/ C ,lg n/ C ,n/ D ,n/.

Así, el paralelismo es ,n3=,n/ D ,n2/. El ejercicio 27.2-3 le pide que paralelice el ciclo interno para obtener un paralelismo de ,n3= lg n/, lo cual no puede hacer directamente usando paralelo porque crearía carreras.

Un algoritmo de subprocesos múltiples de divide y vencerás para la multiplicación de matrices

Como aprendimos en la Sección 4.2, podemos multiplicar nn matrices en serie en el tiempo ..nlg 7/ D O.n2:81/ usando la estrategia divide y vencerás de Strassen, lo que nos motiva a considerar el multiproceso de dicho algoritmo. Comenzamos, como lo hicimos en la Sección 4.2, con subprocesos múltiples de un algoritmo más simple de divide y vencerás.

Recuerde de la página 77 que el procedimiento CUADRADO-MATRIZ-MULTIPLICAR-RECURSIVO, que multiplica dos matrices A y B de nn para producir la matriz C de nn, se basa en dividir cada una de las tres matrices en cuatro submatrices n=2 n=2 :

$$\begin{array}{llll} \text{ANUNCIO} & \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} & \begin{matrix} BD \\ B_{21} & B_{22} \end{matrix} & \begin{matrix} : \\ CD \end{matrix} & \begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{matrix} \\ & ; & & & : \end{array}$$

Entonces, podemos escribir el producto matricial como

$$\begin{array}{llll} \begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{matrix} & \begin{matrix} D \\ : \end{matrix} & \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} & \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix} \\ & & & \\ & & \begin{matrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{matrix} & \begin{matrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{matrix} & (27.6) \end{array}$$

Así, para multiplicar dos matrices de nn, realizamos ocho multiplicaciones de matrices de n=2n=2 y una suma de matrices de nn. El siguiente pseudocódigo implementa

esta estrategia divide y vencerás usando paralelismo anidado. A diferencia del procedimiento MATRIZ CUADRADA -MULTIPLICACIÓN-RECURSIVA en el que se basa, P-MATRIZ MULTIPLICACIÓN-RECURSIVA toma como parámetro la matriz de salida para evitar asignar matrices innecesariamente.

```

P-MATRIZ-MULTIPLICAR-RECURSIVA.C; A; B/
1 n D A:filas 2 si n
== 1 3 c11 D
a11b11 4 si no, sea T una
nueva matriz nn 5 divida A, B, C y T en n=2
n=2 submatrices
      A11; A12; A21; A22; B11; B12; B21; B22; C11; C12; C21;
      C22; y T11; T12; T21; T22;
6      respectivamente generan P-MATRIX-MULTIPLY-
7      RECURSIVE.C11; A11; B11/ spawn P-MATRIX-MULTIPLY-
8      RECURSIVE.C12; A11; B12/ spawn P-MATRIX-MULTIPLY-
9      RECURSIVE.C21; A21; B11/ spawn P-MATRIX-MULTIPLY-
10     RECURSIVE.C22; A21; B12/ spawn P-MATRIX-MULTIPLY-
11     RECURSIVE.T11; A12; B21/ spawn P-MATRIX-MULTIPLY-
12     RECURSIVE.T12; A12; B22/ spawn P-MATRIX-MULTIPLY-
13     RECURSIVE.T21; A22; B21/ P-MATRIZ-MULTIPLICAR-RECURSIVA.T22; A22; B22/
14
15     sincronizar paralelo para i D
diseñado
16     1 a n paralelo para j D 1 a n
17     cij D cij C tij

```

La línea 3 maneja el caso base, donde estamos multiplicando matrices 1 1. Manejamos el caso recursivo en las líneas 4-17. Asignamos una matriz temporal T en la línea 4, y la línea 5 divide cada una de las matrices A, B, C y T en n=2 n=2 submatrices.

(Al igual que con SQUARE-MATRIX-MULTIPLY-RECURSIVE en la página 77, pasamos por alto el problema menor de cómo usar los cálculos de índice para representar secciones de submatriz de una matriz). La llamada recursiva en la línea 6 establece la submatriz C11 en el producto de submatriz A11B11 , de manera que C11 es igual al primero de los dos términos que forman su suma en la ecuación (27.6). De manera similar, las líneas 7 a 9 establecen C12, C21 y C22 en el primero de los dos términos que igualan sus sumas en la ecuación (27.6). La línea 10 establece la submatriz T11 con el producto de submatriz A12B21, de modo que T11 es igual al segundo de los dos términos que forman la suma de C11 . Las líneas 11 a 13 establecen T12, T21 y T22 en el segundo de los dos términos que forman las sumas de C12, C21 y C22, respectivamente. Se generan las primeras siete llamadas recursivas y la última se ejecuta en la hebra principal. La declaración de sincronización en la línea 14 asegura que se hayan calculado todos los productos de la submatriz en las líneas 6-10.

después de lo cual sumamos los productos de T en C usando los bucles for paralelos doblemente anidados en las líneas 15–17.

Primero analizamos el trabajo M1.n/ del procedimiento P-MATRIX-MULTIPLY-RECURSIVE , haciendo eco del análisis de tiempo de ejecución en serie de su progenitor SQUARE MATRIX-MULTIPLY-RECURSIVE. En el caso recursivo, particionamos en ,1/ tiempo, realizamos ocho multiplicaciones recursivas de matrices  $n=2$   $n=2$  y terminamos con el trabajo ,n2/ sumando dos matrices nn. Así, la recurrencia para el trabajo M1.n/ es

$$M1.n/ \leq 8M1.n=2/ + C_{n,n} + D_{n,n}$$

por el caso 1 del teorema maestro. En otras palabras, el trabajo de nuestro algoritmo multiproceso es asintóticamente el mismo que el tiempo de ejecución del procedimiento SQUARE MATRIX-MULTIPLY en la Sección 4.2, con sus bucles triplemente anidados.

Para determinar el intervalo M1.n/ de P-MATRIX-MULTIPLY-RECURSIVE, primero observamos que el intervalo para la partición es ,1/, que está dominado por el intervalo ,lg n/ del bucle for paralelo doblemente anidado en líneas 15-17. Debido a que las ocho llamadas recursivas paralelas se ejecutan en matrices del mismo tamaño, el intervalo máximo para cualquier llamada recursiva es solo el intervalo de cualquiera. Por lo tanto, la recurrencia para el intervalo M1.n/ de P-MATRIX-MULTIPLY-RECURSIVE es

$$M1.n/ \leq M1.n=2/ + C_{lg n} : \quad (27.7)$$

Esta recurrencia no cae bajo ninguno de los casos del teorema maestro, pero cumple la condición del ejercicio 4.6-2. Por el ejercicio 4.6-2, por lo tanto, la solución a la recurrencia (27.7) es  $M1.n/ \leq lg 2 n/$ .

Ahora que conocemos el trabajo y el intervalo de P-MATRIX-MULTIPLICIDAD-RECURSIVA, podemos calcular su paralelismo como  $M1.n/ = M1.n/ \leq lg 2 n/$ , que es muy alto.

### Método de Strassen de subprocesos múltiples

Para el algoritmo de Strassen multiproceso, seguimos el mismo esquema general que en la página 79, solo usando paralelismo anidado:

1. Divida las matrices de entrada A y B y la matriz de salida C en  $n=2$   $n=2$  submatrices, como en la ecuación (27.6). Este paso toma ,1/ trabajo y span por cálculo de índice.
2. Crear 10 matrices S1; S2;:::;S10, cada una de las cuales es  $n=2$   $n=2$  y es la suma o diferencia de dos matrices creadas en el paso 1. Podemos crear las 10 matrices con ,n2/ trabajo y ,lg n/ span mediante el uso de bucles for paralelos doblemente anidados .

3. Utilizando las submatrices creadas en el paso 1 y las 10 matrices creadas en el paso 2, genere recursivamente el cálculo de siete productos de matriz  $n=2$   $n=2$   $P_1; P_2; \dots; P_7$ .
4. Calcular las submatrices deseadas  $C_{11}; C_{12}; C_{21}; C_{22}$  de la matriz de resultado  $C$  sumando y restando varias combinaciones de las matrices  $P_i$ , una vez más usando bucles for paralelos doblemente anidados . Podemos calcular las cuatro submatrices con  $.n^2/\log n/\text{span}$ .

Para analizar este algoritmo, primero observamos que dado que la serialización es la misma que el algoritmo serial original, el trabajo es solo el tiempo de ejecución de la serialización, a saber,  $\log 7^n$  . En cuanto a P-MATRIX-MULTIPLY-RECURSIVE, podemos idear una recurrencia para el lapso. En este caso, siete llamadas recursivas se ejecutan en paralelo, pero como todas operan sobre matrices del mismo tamaño, obtenemos la misma recurrencia (27.7) que obtuvimos para P-MATRIZ-MULTIPLICAR-RECURSIVA, que tiene solución  $\log 7^n$  . Por lo tanto, el paralelismo del método de Strassen de subprocessos múltiples es  $\log 7^n = \log 2^n$ , que es alto, aunque ligeramente menor que el paralelismo de P-MATRIX-MULTIPLY-RECURSIVE.

### Ejercicios

#### 27.2-1

Dibuje el dag de cálculo para calcular P-SQUARE-MATRIX-MULTIPLY en 22 matrices, etiquetando cómo los vértices en su diagrama corresponden a hebras en la ejecución del algoritmo. Utilice la convención de que los bordes de generación y llamada apuntan hacia abajo, los bordes de continuación apuntan horizontalmente hacia la derecha y los bordes de retorno apuntan hacia arriba. Suponiendo que cada hebra toma una unidad de tiempo, analice el trabajo, el lapso y el paralelismo de este cálculo.

#### 27.2-2

Repita el ejercicio 27.2-1 para P-MATRIZ-MULTIPLICAR-RECURSIVA.

#### 27.2-3

Proporcione un pseudocódigo para un algoritmo de subprocessos múltiples que multiplica dos matrices  $n \times n$  con trabajo  $n^3$  pero abarca solo  $\log n$ . Analiza tu algoritmo.

#### 27.2-4

Proporcione un pseudocódigo para un algoritmo multihebra eficiente que multiplique la matriz  $a \times p$  por la matriz  $q \times r$ . Su algoritmo debe ser altamente paralelo incluso si cualquiera de  $p, q$  y  $r$  son 1. Analice su algoritmo.

## 27.2-5

Proporcione un pseudocódigo para un algoritmo multiproceso eficiente que transponga una matriz  $n \times n$  en su lugar mediante el uso de divide y vencerás para dividir la matriz recursivamente en cuatro submatrices  $n=2^{k-1}$ . Analiza tu algoritmo.

## 27.2-6

Proporcione un pseudocódigo para una implementación eficiente de subprocessos múltiples del algoritmo Floyd Warshall (consulte la Sección 25.2), que calcula las rutas más cortas entre todos los pares de vértices en un gráfico de borde ponderado. Analiza tu algoritmo.

## 27.3 Ordenación por fusión de subprocessos múltiples

Vimos por primera vez la ordenación por fusión en serie en la Sección 2.3.1, y en la Sección 2.3.2 analizamos su tiempo de ejecución y mostramos que era  $\sim n \lg n$ . Debido a que la ordenación por fusión ya usa el paradigma divide y vencerás, parece un excelente candidato para subprocessos múltiples usando paralelismo anidado.

Podemos modificar fácilmente el pseudocódigo para que se genere la primera llamada recursiva:

```

MERGE-SORT0 .A; pag; r/ 1 si
  p<r q D pb C r/
    =2c 2 genera MERGE-
      SORT0 .A; pag; q/ MERGE-SORT0 .A; q C 1; r/
      4
      5
      6      sincronizar MERGE.A; pag; q; r/

```

Al igual que su homólogo en serie, MERGE-SORT0 ordena el subarreglo  $A[0:p-1]$ . Después de que se hayan completado las dos subrutinas recursivas en las líneas 3 y 4, lo cual está garantizado por la declaración de sincronización en la línea 5, MERGE-SORT0 llama al mismo procedimiento MERGE que en la página 31.

Analicemos MERGE-SORT0 . Para hacerlo, primero debemos analizar MERGE. Recuerde que su tiempo de ejecución en serie para fusionar  $n$  elementos es  $\sim n$ . Debido a que MERGE es serial, tanto su trabajo como su duración son  $\sim n$ . Así, la siguiente recurrencia caracteriza el trabajo MS0 1.n/ de MERGE-SORT0 sobre  $n$  elementos:

MS0 1.n/ D 2 MS0 1.n=2/ C ,n/ D ,n largo n/ ;

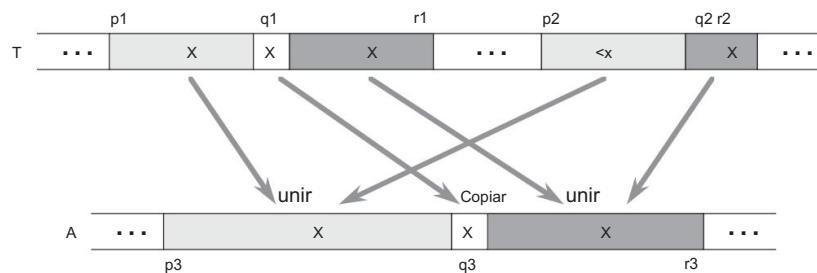


Figura 27.6 La idea detrás de la fusión de subprocessos múltiples de dos subarreglos ordenados  $T \in p_1 :: r_1$  y  $T \in p_2 :: r_2$  en el subarreglo  $A \in p_3 :: r_3$ . Si  $x \in T \in q_1$  es la mediana de  $T \in p_1 :: r_1$  y  $q_2$  es el lugar en  $T \in p_2 :: r_2$  tal que  $x$  estaría entre  $T \in q_1$  y  $T \in q_2$ , cada elemento en los subarreglos  $T \in p_1 :: q_1$  y  $T \in p_2 :: q_2$  (sombreado claro) es menor o igual que  $x$ , y cada elemento en los subarreglos  $T \in q_1 C 1 :: r_1$  y  $T \in q_2 C 1 :: r_2$  (muy sombreado) es al menos  $x$ . Para combinar, calculamos el índice  $q_3$  donde  $x$  pertenece a  $A \in p_3 :: r_3$ , copiamos  $x$  en  $A \in q_3$  y luego combinamos recursivamente  $T \in p_1 :: q_1$  con  $T \in p_2 :: q_2$  en  $A \in p_3 :: q_3$  y  $T \in q_1 C 1 :: r_1$  con  $T \in q_2 C 1 :: r_2$  en  $A \in q_3 C 1 :: r_3$ .

que es lo mismo que el tiempo de ejecución en serie de la ordenación por fusión. Dado que las dos llamadas recursivas de MERGE-SORT0 pueden ejecutarse en paralelo, el intervalo<sub>MS0</sub> viene dado por la recurrencia

$$MS0[1..n] / D = MS0[1..n-2] / C + MS0[n-1..n] / D$$

Por lo tanto, el paralelismo de MERGE-SORT0 llega a  $MS0[1..n] = MS0[1..n] / D \cdot \lg n$ , que es una cantidad impresionante de paralelismo. Para ordenar 10 millones de elementos, por ejemplo, podría lograr una aceleración lineal en unos pocos procesadores, pero no escalaría de manera efectiva a cientos de procesadores.

Probablemente ya haya averiguado dónde está el cuello de botella del paralelismo en este tipo de combinación de subprocessos múltiples: el procedimiento MERGE en serie. Aunque inicialmente la fusión puede parecer inherentemente serial, podemos, de hecho, crear una versión multiproceso de la misma mediante el uso de paralelismo anidado.

Nuestra estrategia divide y vencerás para la fusión de subprocessos múltiples, que se ilustra en la figura 27.6, opera en subarreglos de un arreglo T. Supongamos que fusionamos los dos subarreglos ordenados  $T \in p_1 :: r_1$  de longitud  $n_1$  y  $T \in p_2 :: r_2$  de longitud  $n_2$  en otro subarreglo  $A \in p_3 :: r_3$ , de longitud  $n_3$ . Sin pérdida de generalidad, hacemos la suposición simplificada de que  $n_1 \leq n_2$ .

Primero encontramos el elemento medio  $x \in T \in q_1$  del subarreglo  $T \in p_1 :: r_1$ , donde  $q_1 \leq b \leq p_1 + r_1 - 1 = 2c$ . Debido a que el subarreglo está ordenado,  $x$  es una mediana de  $T \in p_1 :: r_1$ : cada elemento en  $T \in p_1 :: q_1$  no es mayor que  $x$ , y cada elemento en  $T \in q_1 C 1 :: r_1$  no es menor que  $x$ . Luego usamos la búsqueda binaria para encontrar el

indexar q2 en el subarreglo  $T \in p2 :: r2$  de modo que el subarreglo aún estaría ordenado si insertáramos x entre  $T \in q2 1$  y  $T \in q2$ .

Luego fusionamos los subarreglos originales  $T \in p1 :: r1$  y  $T \in p2 :: r2$  en  $A \in p3 :: r3$  de la siguiente manera:

1. Ajuste  $q3 D p3 C .q1 p1 / C .q2 p2 /$ .
2. Copie x en  $A \in q3$ .
3. Combine recursivamente  $T \in p1 :: q1 1$  con  $T \in p2 :: q2 1$ , y coloque el resultado en el subarreglo  $A \in p3 :: q3 1$ .
4. Combine recursivamente  $T \in q1 C 1 :: r1$  con  $T \in q2 :: r2$  y coloque el resultado en el subarreglo  $A \in q3 C 1 :: r3$ .

Cuando calculamos q3, la cantidad  $q1p1$  es el número de elementos en el subarreglo  $T \in p1 :: q1 1$ , y la cantidad  $q2 p2$  es el número de elementos en el subarreglo  $T \in p2 :: q2 1$ . Por lo tanto, su suma es el número de elementos que terminan antes de x en el subarreglo  $A \in p3 :: r3$ .

El caso base ocurre cuando  $n1 D n2 D 0$ , en cuyo caso no tenemos trabajo que hacer para fusionar los dos subarreglos vacíos. Dado que hemos supuesto que el subconjunto  $T \in p1 :: r1$  es al menos tan largo como  $T \in p2 :: r2$ , es decir,  $n1 \geq n2$ , podemos comprobar el caso base simplemente comprobando si  $n1 = 0$ . También debemos asegurarnos de que la recursividad maneja correctamente el caso cuando solo uno de los dos subarreglos está vacío, que, según nuestra suposición de que  $n1 \geq n2$ , debe ser el subarreglo  $T \in p2 :: r2$ .

Ahora, pongamos estas ideas en pseudocódigo. Comenzamos con la búsqueda binaria, que expresamos en serie. El procedimiento BINARIO-BUSQUEDA.x; T; pag; r/ toma una clave x y un subarreglo  $T \in p :: r$ , y devuelve uno de los siguientes:

Si  $T \in p :: r$  está vacío ( $r < p$ ), entonces devuelve el índice p.

Si  $x \in T \in p$ , y por lo tanto menor o igual que todos los elementos de  $T \in p :: r$ , entonces devuelve el índice p.

Si  $x > T \in p$ , entonces devuelve el mayor índice q en el rango  $p < q \leq C 1$  tal que  $T \in q 1 < x$ .

Aquí está el pseudocódigo:

```
BINARIO-BUSQUEDA.x; T; pag;
r/ 1 baja D p 2
alta D max.p; r C 1 / 3 while
grave < alto 4 mid D
b.low C high/=2c if x T \in mid high D mid
5      else low D mid
6      C 1 8 return
7      high
```

La llamada BINARIO-SEARCH.x; T; pag; r/ toma  $\lg n$  tiempo de serie en el peor de los casos, donde  $n \leq r \leq C$  1 es el tamaño del subarreglo en el que se ejecuta. (Consulte el ejercicio 2.3-5.) Dado que BÚSQUEDA-BINARIA es un procedimiento en serie, su trabajo y intervalo en el peor de los casos son ambos  $\lg n$ .

Ahora estamos preparados para escribir pseudocódigo para el propio procedimiento de fusión multiproceso. Al igual que el procedimiento MERGE de la página 31, el procedimiento P-MERGE asume que los dos subarreglos que se fusionarán se encuentran dentro del mismo arreglo. Sin embargo, a diferencia de MERGE, P-MERGE no asume que los dos subarreglos que se fusionarán son adyacentes dentro del arreglo. (Es decir, P-MERGE no requiere que  $p_2 \leq r_1 < C$  1.) Otra diferencia entre MERGE y P-MERGE es que P-MERGE toma como argumento un subarreglo de salida A en el que se deben almacenar los valores combinados. La llamada P-MERGE.T; p1; r1; p2; r2; A; p3/ fusiona los subarreglos ordenados T  $\leq p_1 :: r_1$  y T  $\leq p_2 :: r_2$  en el subarreglo A  $\leq p_3 :: r_3$ , donde  $r_3 \leq p_3 < C$  . $r_1 \leq p_1 < C$  . $r_2 \leq p_2 < C$  . $r_3 \leq p_3 < C$  . $r_1 \leq p_1 < C$  . $r_2 \leq p_2 < C$  1 y no se proporciona como entrada.

```

P-FUSIÓN.T; p1; r1; p2; r2; A; p3/ 1 n1
D r1 p1 C 1 2 n2 D r2 p2
C 1 3 si n1 < n2 4
    intercambia p1           // asegurar que n1 n2
    con p2 intercambia r1 con r2
    5      intercambia n1 con n2
    6      7 si n1 == 0 8 regresa 9
    sino q1 D b.p1           // ambos vacíos?
    C r1/=2c q2 D
    BÚSQUEDA-BINARIA.T  $\leq q_1$ ; T;
    p2; r2/ 10 q3 D p3 C .q1 p1/ C .q2 p2/ A $\leq q_3$  DT
    11  $\leq q_1$  spawn P-FUSIÓN.T; p1; q1 1; p2; q2 1; A; p3/
    12      P-FUSIÓN.T; q1
    13      C 1; r1; q2; r2; A; q3 C 1/ sincronización
    14
    15

```

El procedimiento P-MERGE funciona de la siguiente manera. Las líneas 1 y 2 calculan las longitudes  $n_1$  y  $n_2$  de los subarreglos T  $\leq p_1 :: r_1$  y T  $\leq p_2 :: r_2$ , respectivamente. Las líneas 3 a 6 imponen la suposición de que  $n_1 \leq n_2$ . La línea 7 comprueba el caso base, donde el subarreglo T  $\leq p_1 :: r_1$  está vacío (y por lo tanto T  $\leq p_2 :: r_2$ ), en cuyo caso simplemente regresamos. Las líneas 9 a 15 implementan la estrategia divide y vencerás. La línea 9 calcula el punto medio de T  $\leq p_1 :: r_1$ , y la línea 10 encuentra el punto  $q_2$  en T  $\leq p_2 :: r_2$  tal que todos los elementos en T  $\leq p_2 :: q_2$  1 son menores que T  $\leq q_1$  (que corresponde a  $x$ ) y todos los elementos en T  $\leq q_2 :: p_2$  son al menos tan grandes como T  $\leq q_1$ . Línea 11 com.

pone el índice q3 del elemento que divide el subarreglo de salida AŒp3 ::r3 en AŒp3 ::q3 1 y AŒq3C1::r3, y luego la línea 12 copia T Œq1 directamente en AŒq3.

Luego recurrimos usando paralelismo anidado. La línea 13 genera el primer subproblema, mientras que la línea 14 llama al segundo subproblema en paralelo. La declaración de sincronización en la línea 15 asegura que los subproblemas se hayan completado antes de que regrese el procedimiento. (Dado que cada procedimiento ejecuta implícitamente una sincronización antes de regresar, podríamos haber omitido la declaración de sincronización en la línea 15, pero incluirla es una buena práctica de codificación). vacío, el código funciona correctamente. La forma en que funciona es que en cada llamada recursiva, un elemento mediano de T Œp1 ::r1 se coloca en el subarreglo de salida, hasta que T Œp1 ::r1 finalmente se vacía, activando el caso base.

#### Análisis de fusión multiproceso

Primero derivamos una recurrencia para el tramo PM1.n/ de P-MERGE, donde los dos subarreglos contienen un total de  $n$  D  $n1Cn2$  elementos. Debido a que el engendro en la línea 13 y la llamada en la línea 14 operan lógicamente en paralelo, necesitamos examinar solo la más costosa de las dos llamadas. La clave es entender que en el peor de los casos, el número máximo de elementos en cualquiera de las llamadas recursivas puede ser como máximo  $3n=4$ , lo que vemos a continuación. Como las líneas 3 a 6 aseguran que  $n2 \leq n1$ , se sigue que  $n2 \leq 2n2=2 \cdot n1 \leq n2/2 \leq D \leq n=2$ . En el peor de los casos, una de las dos llamadas recursivas fusiona  $bn1=2c$  elementos de  $T \ Œp1 ::r1$  con todos los  $n2$  elementos de  $T \ Œp2 ::r2$  y, por lo tanto, el número de elementos involucrados en la llamada es

$$\begin{aligned} bn1=2c & C n2 & n1=2 & C n2=2 & C n2=2 \\ & C .n1 & C n2/2 & C n2=2 & n=2 \\ & C n=4 & D \\ 3n=4 : & & & & \end{aligned}$$

Agregando el costo  $\lg n$  de la llamada a BINARY-SEARCH en la línea 10, obtenemos la siguiente recurrencia para el intervalo del peor de los

casos:  $PM1.n/ \leq PM1.3n=4/ C \lg n/$  : (27.8)

(Para el caso base, el lapso es  $\lg 1/ = 1/$ , ya que las líneas 1 a 8 se ejecutan en tiempo constante). Esta recurrencia no cae bajo ninguno de los casos del teorema maestro, pero cumple la condición del Ejercicio 4.6-2. Por tanto, la solución a la recurrencia (27.8) es  $PM1.n/ \leq \lg 2 n/$ .

Ahora analizamos el trabajo PM1.n/ de P-MERGE sobre  $n$  elementos, que resulta ser  $\lg n/$ . Dado que cada uno de los  $n$  elementos debe copiarse del arreglo  $T$  al arreglo  $A$ , tenemos  $PM1.n/ \leq n/$ . Por lo tanto, solo queda demostrar que  $PM1.n/ \leq On/$ .

Primero derivaremos una recurrencia para el trabajo del peor de los casos. La búsqueda binaria en la línea 10 cuesta  $\lg n/$  en el peor de los casos, que domina el otro trabajo fuera

de las llamadas recursivas. Para las llamadas recursivas, observe que aunque las llamadas recursivas en las líneas 13 y 14 pueden fusionar diferentes números de elementos, juntas las dos llamadas recursivas fusionan como máximo  $n$  elementos (en realidad  $n \geq 1$  elementos, ya que  $T \leq q_1$  no participa en ninguna llamada recursiva). Además, como vimos al analizar el lento, una llamada recursiva opera en un máximo de  $3n=4$  elementos. Obtenemos por tanto la reaparición

$$PM1.n / D PM1..n / C PM1..1 ..n / C O.lg n / ; \quad (27.9)$$

donde  $\_$  se encuentra en el rango  $1=4 \dots 3=4$ , y donde entendemos que el valor real de  $\_$  puede variar para cada nivel de recursividad.

Probamos que la recurrencia (27.9) tiene solución  $PM1.D = O(n \lg n)$  por el método de sustitución.

Supongamos que  $PM1..n \leq c_1 n c_2 \lg n$  para algunas constantes positivas  $c_1$  y  $c_2$ .

Sustituyendo nos da

$$\begin{aligned} PM1..n & \leq c_1 n c_2 \lg ..n / C ..c_1 ..1 ..n / C ..c_2 \lg ..1 ..n / C ..\lg n \\ & \leq c_1 ..C ..1 ..n / C ..c_2 \lg ..n / C ..\lg ..1 ..n / C ..\lg n / D ..c_1 n c_2 \lg ..C ..\lg n C ..\lg ..1 ..C ..\lg n \\ & \leq c_1 ..C ..\lg n / D ..c_1 n c_2 \lg n ..c_2 \lg n C ..\lg ..1 ..C ..\lg n / C ..\lg n / c_1 n c_2 \lg n \end{aligned}$$

:

ya que podemos elegir  $c_2$  lo suficientemente grande como para que  $c_2 \lg n \leq C ..\lg ..1 ..C ..\lg n$  domine el término  $\lg n$ . Además, podemos elegir  $c_1$  lo suficientemente grande como para satisfacer las condiciones base de la recurrencia. Dado que el trabajo  $PM1..n$  de P-MERGE es tanto  $O(n \lg n)$  como  $O(n)$ , tenemos  $PM1..n \leq O(n \lg n)$ .

El paralelismo de P-MERGE es  $PM1..n = O(n \lg n)$ .

#### Ordenación por fusión multiproceso

Ahora que tenemos un procedimiento de fusión de subprocessos múltiples muy bien paralelizado, podemos incorporarlo en una ordenación de fusión de subprocessos múltiples. Esta versión de ordenación por fusión es similar al procedimiento MERGE-SORT0 que vimos anteriormente, pero a diferencia de MERGE-SORT0 como argumento, un subarray de salida  $B$ , que contendrá el resultado ordenado. En particular, la llamada  $P-MERGE-SORT(A; pag; r; B; s; p)$  ordena los elementos en  $A[0:p-1]$  y los almacena en  $B[s:p]$ .

```

P-COMBINAR-ORDENAR.A; pag; r;
B; s/ 1 n D rp C 1 2 if
n == 1 3 BŒs
D AŒp 4 else sea T
Œ1 : : n sea un nuevo arreglo 5 q D bp C
r/=2c 6 q0 D qp C 1 7 spawn
P-MERGE -SORT.A; pag; q;
T; 1 / 8 P-COMBINACIÓN-CLASIFICACIÓN.A; q C 1; r;
T; q0 C 1/ sincronización
9
10      P-FUSIÓN.T; 1; q0 ; q0 C 1; norte; B; s/

```

Después de que la línea 1 calcule el número n de elementos en el subarreglo de entrada AŒp : : r, las líneas 2–3 manejan el caso base cuando el arreglo tiene solo 1 elemento. Las líneas 4 a 6 se configuran para la generación recursiva en la línea 7 y la llamada en la línea 8, que funcionan en paralelo. En particular, la línea 4 asigna una matriz temporal T con n elementos para almacenar los resultados de la ordenación de combinación recursiva. La línea 5 calcula el índice q de AŒp : : r para dividir los elementos en los dos subarreglos AŒp : : q y AŒq C 1::r que serán ordenados recursivamente, y la línea 6 continúa para calcular el número q0 de elementos en el primer subarreglo AŒp : : q, que la línea 8 usa para determinar el índice inicial en T de dónde almacenar el resultado ordenado de AŒq C 1::r. En ese punto, se realizan la generación y la llamada recursiva, seguidas de la sincronización en la línea 9, lo que obliga al procedimiento a esperar hasta que finalice el procedimiento generado. Finalmente, la línea 10 llama a P-MERGE para fusionar los subarreglos ordenados, ahora en T Œ1 : : q0 y T Œq0 C 1 : : n, en el subarreglo de salida BŒs : : s C r p.

#### Análisis de clasificación de fusión multiproceso

Comenzamos analizando el trabajo PMS1.n/ de P-MERGE-SORT, que es considerablemente más fácil que analizar el trabajo de P-MERGE. En efecto, el trabajo está dado por el reaparición

PMS1.n/ D 2 PMS1.n=2/ C PM1.n/

D 2 PMS1.n=2/ C ,n/ :

Esta recurrencia es la misma que la recurrencia (4.4) para MERGE-SORT ordinario de la Sección 2.3.1 y tiene solución PMS1.n/ D ,n lg n/ por el caso 2 del maestro teorema.

Ahora derivamos y analizamos una recurrencia para el peor de los casos PMS1.n/. Debido a que las dos llamadas recursivas a P-MERGE-SORT en las líneas 7 y 8 operan lógicamente en paralelo, podemos ignorar una de ellas, obteniendo la recurrencia

PMS1.n/ D PMS1.n=2/ C PM1.n / D PMS1.n=2/ C

$$\dots \lg 2 n : \quad (27.10)$$

En cuanto a la recurrencia (27.8), el teorema maestro no se aplica a la recurrencia (27.10), pero el ejercicio 4.6-2 sí. La solución es  $PMS1.n/ D \dots \lg 3 n/$ , por lo que el intervalo de P-MERGE-SORT es  $\dots \lg 3 n/$ .

La combinación paralela le da a P-MERGE-SORT una ventaja de paralelismo significativa sobre MERGE-SORT0 . Recuerde que el paralelismo del procedimiento MERGE- , que llama el se SORT0 rial MERGE es solo  $\dots \lg n/$ . Para P-MERGE-SORT, el paralelismo es

$PMS1.n/=PMS1.n/ D \dots n \lg n/=\dots \lg 3 n/ D \dots n= \lg 2 n/ ;$

que es mucho mejor tanto en la teoría como en la práctica. Una buena implementación en la práctica sacrificaría algo de paralelismo al toscar el caso base para reducir las constantes ocultas por la notación asintótica. La forma sencilla de hacer más grueso el caso base es cambiar a una ordenación en serie normal, tal vez una ordenación rápida, cuando el tamaño de la matriz es lo suficientemente pequeño.

## Ejercicios

### 27.3-1

Explique cómo engrosar el caso base de P-MERGE.

### 27.3-2

En lugar de encontrar un elemento mediano en el subarreglo más grande, como lo hace P-MERGE , considere una variante que encuentre un elemento mediano de todos los elementos en los dos ordenados. subarreglos usando el resultado del Ejercicio 9.3-8. Proporcione un pseudocódigo para un procedimiento de fusión de subprocessos múltiples eficiente que utilice este procedimiento de búsqueda de la mediana. Analice su algoritmo.

### 27.3-3

Proporcione un algoritmo multiproceso eficiente para particionar un arreglo alrededor de un pivote, como se hace con el procedimiento de PARTICIÓN en la página 171. No necesita particionar el arreglo en su lugar. Haz que tu algoritmo sea lo más paralelo posible. Analiza tu algoritmo.

(Sugerencia: es posible que necesite una matriz auxiliar y que deba realizar más de una pasada sobre los elementos de entrada).

### 27.3-4

Proporcione una versión multiproceso de RECURSIVE-FFT en la página 911. Haga su implementación lo más paralela posible. Analiza tu algoritmo.

## 27.3-5 ?

Proporcione una versión multiproceso de RANDOMIZED-SELECT en la página 216. Haga que su implementación sea lo más paralela posible. Analiza tu algoritmo. (Sugerencia: use el algoritmo de partición del ejercicio 27.3-3).

## 27.3-6 ?

Muestre cómo SELECT multiproceso de la Sección 9.3. Haga su implementación lo más paralela posible. Analiza tu algoritmo.

## Problemas

## 27-1 Implementación de bucles paralelos usando paralelismo anidado

Considere el siguiente algoritmo multiproceso para realizar sumas por pares en arreglos de  $n$  elementos  $A[0:n], B[0:n]$ , almacenando las sumas en  $C[0:n]$ .

norte:

SUMA-ARRAYS.A; B; C / 1

paralelo para  $i \in D$  1 a  $A[i]:longitud$

$C[i] = A[i] + B[i]$

- Vuelva a escribir el ciclo paralelo en SUM-ARRAYS utilizando el paralelismo anidado (generación y sincronización) a la manera de MAT-VEC-MAIN-LOOP. Analice el paralelismo de su implementación.

Considere la siguiente implementación alternativa del bucle paralelo, que contiene un valor de tamaño de grano que debe especificarse:

SUMA-ARRAYS0 .A; B; C / 1

$n \in D$   $A[i]:longitud$  2

tamaño de grano  $D \in 3$  // por determinar

$r \in D$   $d_n = \text{tamaño de grano}$

4 para  $k \in D$  0 a  $r - 1$

generar ADD-SUBARRRAY.A; B; C;  $k$  tamaño de grano C 1;

min..k C 1 / tamaño de grano; norte//

6 sincronización

ADD-SUBARRRAY.A; B; C;  $i; j / 1$

para  $k \in D$   $i \leq j$   $C[k] = A[i] + B[j]$

$D \in A[k] C B[k]$

b. Supongamos que establecemos un tamaño de grano D 1. ¿Cuál es el paralelismo de este implemento? tación?

C. Proporcione una fórmula para el lapso de SUM-ARRAYS0 en términos de n y tamaño de grano.

Obtenga el mejor valor para el tamaño de grano para maximizar el paralelismo.

#### 27-2 Ahorro de espacio temporal en la multiplicación de matrices El

procedimiento P-MATRIX-MULTIPLY-RECURSIVE tiene la desventaja de que debe asignar una matriz temporal T de tamaño nn, lo que puede afectar negativamente a las constantes ocultas por la notación ,. Sin embargo, el procedimiento P-MATRIX-MULTIPLY-RECURSIVE tiene un alto paralelismo. Por ejemplo, ignorando las constantes en la notación , el paralelismo para multiplicar 1000 1000 matrices llega a aproximadamente  $1000^3 = 10^9$  D 107, ya que  $\lg 1000 = 10$ . La mayoría de las computadoras paralelas tienen mucho menos de 10 millones de procesadores.

a. Describa un algoritmo recursivo de subprocessos múltiples que elimine la necesidad de la matriz temporal T a costa de aumentar el intervalo a ,n/. (Sugerencia: calcule CDCC AB siguiendo la estrategia general de P-MATRIX-MULTIPLY RECURSIVE, pero inicialice C en paralelo e inserte una sincronización en una ubicación elegida juiciosamente).

b. Proporcione y resuelva recurrencias para el trabajo y el lapso de su implementación.

C. Analice el paralelismo de su implementación. Ignorando las constantes en la notación , estime el paralelismo en matrices 1000 1000. Compare con el paralelismo de P-MATRIX-MULTIPLY-RECURSIVE.

#### 27-3 Algoritmos matriciales de subprocessos múltiples

a. Paralelice el procedimiento LU-DECOMPOSITION en la página 821 proporcionando un pseudocódigo para una versión de subprocessos múltiples de este algoritmo. Haga que su implementación sea lo más paralela posible y analice su trabajo, extensión y paralelismo.

b. Haga lo mismo para DESCOMPOSICIÓN LUP en la página 824.

C. Haga lo mismo para LUP-SOLVE en la página 817.

d. Haga lo mismo para un algoritmo de subprocessos múltiples basado en la ecuación (28.13) para en Verting una matriz definida positiva simétrica.

## 27-4 Reducciones de subprocessos múltiples y cálculos de prefijos

Una reducción “ de un arreglo  $x[1:n]$ , donde “ es un operador asociativo, es el valor

$y = x[1] \text{ op } x[2] \dots \text{ op } x[n]$

El siguiente procedimiento calcula la reducción “ de un subarreglo  $x[i:j]$  en serie.

```
REDUCIR.x; i; j /  
1 y D x[i:j]  
para k D i C 1 a j y D y  
3      x[k] 4 regresa  
y
```

- a. Utilice el paralelismo anidado para implementar un algoritmo multiproceso P-REDUCE, que realiza la misma función con „n/ work y „lg n/ span. Analiza tu algoritmo.

Un problema relacionado es el de calcular un prefijo “, a veces llamado escaneo “, en un arreglo  $x[1:n]$ , donde “ es una vez más un operador asociativo. El escaneo “ produce la matriz  $y[1:n]$  dada por

```
y[1] = x[1];  
y[2] = x[1] “ x[2]; y[3] =  
x[1] “ x[2] “ x[3];  
⋮  
y[n] = x[1] “ x[2] “ x[3] “ … “ x[n];
```

es decir, todos los prefijos del arreglo  $x$  se “sumaron” usando el operador “. El siguiente procedimiento en serie SCAN realiza un cálculo de prefijo “:

```
SCAN.x/  
1 n D x:longitud 2  
sea y[1:n] un nuevo arreglo 3  
y[1] = x[1];  
para i D 2 a n 5 y[i] =  
x[1] “ x[2] “ … “ x[i] 6 devuelve y
```

Desafortunadamente, SCAN multiproceso no es sencillo. Por ejemplo, cambiar el bucle for a un bucle for paralelo crearía carreras, ya que cada iteración del cuerpo del bucle depende de la iteración anterior. El siguiente procedimiento P-SCAN-1 realiza el cálculo del prefijo “ en paralelo, aunque de manera ineficiente:

P-SCAN-1.x/ 1

```
n D x:longitud 2 sea
yŒ1 : : n un nuevo arreglo 3 P-
SCAN-1-AUX.x; y; 1; n/ 4 vuelta y
```

P-ESCANEAR-1-AUX.x; y; i; j /

```
1 paralelo para l D i a j 2 yŒl
D P-REDUCE.x; 1; l/
```

b. Analice el trabajo, el alcance y el paralelismo de P-SCAN-1.

Mediante el uso de paralelismo anidado, podemos obtener un cálculo de prefijo " más eficiente:

P-SCAN-2.x/ 1

```
n D x:longitud 2 sea
yŒ1 : : n un nuevo arreglo 3 P-
SCAN-2-AUX.x; y; 1; n/ 4 vuelta y
```

P-SCAN-2-AUX.x; y; i; j / 1 if i

```
= = j yŒi D
xŒi 2 3 else
k D bi C j /=2c genera P-
4      SCAN-2-AUX.x; y; i; k/ P-SCAN-2-
5      AUX.x; y; k C 1; j /
6
7      sincronización paralela para l D
8          k C 1 a j yŒl D yŒk " yŒl
```

C. Argumente que P-SCAN-2 es correcto y analice su trabajo, amplitud y paralelismo.

Podemos mejorar tanto P-SCAN-1 como P-SCAN-2 realizando el cálculo del prefijo " en dos pasos distintos sobre los datos. En el primer paso, reunimos los términos de varios subarreglos contiguos de x en un arreglo temporal t, y en el segundo paso usamos los términos en t para calcular el resultado final y. El siguiente pseudocódigo implementa esta estrategia, pero se han omitido ciertas expresiones:

P-SCAN-3.x/ 1

```
n D x:longitud 2
sean yŒ1 : : n y tŒ1 : : n nuevos arreglos 3 yŒ1
D xŒ1 4 si n>1 5
P-SCAN-
UP.x; t; 2; n/ 6 P-SCAN-DOWN.xŒ1;
X; t; y; 2; n/ 7 vuelta y
```

P-SCAN-UP.x; t; i; j / 1 if i

```
== j return
xŒi 2 3 else
4 k D bi
C j /=2c 5 tŒk D spawn P-
SCAN-UP.x; t; i; k/ 6 derecha D P-SCAN-UP.x; t; k C
1; j / 7 8
retorno
de sincronización _____ // llenar el espacio en blanco
```

P-ESCANEAR-ABAJO.; X; t; y; i; j /

```
1 si i == j yŒi
D " xŒi 2
3 más
4 K D bi C j /=2c
5 genera P-SCAN-DOWN. ; x;t; y; i; k// complete el espacio en blanco P-SCAN-
6 DOWN. ; x;t; y; k C 1;j// llenar el espacio en blanco
7 sincronizar
```

d. Complete las tres expresiones que faltan en la línea 8 de P-SCAN-UP y las líneas 5 y 6 de P-SCAN-DOWN. Argumente que con las expresiones que proporcionó, P-SCAN-3 es correcta. (Sugerencia: Demuestre que el valor pasado a P-SCAN-DOWN.; x; t; y; i; j / satisface D xŒ1 " xŒ2 "" xŒi 1.)

mi. Analice el trabajo, el alcance y el paralelismo de P-SCAN-3.

27-5 Multiprocesamiento de un cálculo de plantilla simple La ciencia computacional está repleta de algoritmos que requieren que las entradas de una matriz se llenen con valores que dependen de los valores de ciertas entradas vecinas ya calculadas, junto con otra información que no cambia a lo largo del curso del cómputo. El patrón de las entradas vecinas no cambia durante el cálculo y se denomina plantilla. Por ejemplo, la Sección 15.4 presenta

un algoritmo de plantilla para calcular una subsecuencia común más larga, donde el valor en la entrada  $c[i][j]$  depende solo de los valores en  $c[i-1][j], c[i][j-1]$  y  $c[i-1][j-1]$ , así como los elementos  $x_i$  y  $y_j$  dentro de las dos secuencias dadas como entradas. Las secuencias de entrada son fijas, pero el algoritmo completa la matriz bidimensional  $c$  para que calcule la entrada  $c[i][j]$  después de calcular las tres entradas  $c[i-1][j], c[i][j-1]$  y  $c[i-1][j-1]$ .

En este problema, examinamos cómo usar el paralelismo anidado para realizar múltiples subprocessos en un cálculo de plantilla simple en una matriz  $A$  de  $nn$  en la que, de los valores en  $A$ , el valor colocado en la entrada  $A[i][j]$  depende solo de valores en  $A[i-1][j], A[i][j-1]$  y  $A[i-1][j-1]$  (donde  $i \geq 0$  y  $j \geq 0$  supuesto,  $i \neq i_0 \wedge j \neq j_0$ ). En otras palabras, el valor en las entradas  $i$  y  $j$  depende de los valores de las entradas que están arriba y/o a su izquierda, junto con la información estática fuera de la matriz. Además, suponemos a lo largo de este problema que una vez que hemos completado las entradas en las que  $A[i][j]$  depende, podemos completar  $A[i][j]$  en  $\frac{1}{n^2}$  tiempo (como en el procedimiento LCS-LENGTH de la Sección 15.4).

Podemos dividir la matriz  $A$  de  $nn$  en cuatro subarreglos  $n=2$   $n=2$  de la siguiente manera:

$$\begin{array}{cc} \text{ANUNCIO} & \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} \end{array} \quad (27.11)$$

Observe ahora que podemos llenar el subarreglo  $A_{11}$  recursivamente, ya que no depende de las entradas de los otros tres subarreglos. Una vez completado  $A_{11}$ , podemos seguir rellenando  $A_{12}$  y  $A_{21}$  recursivamente en paralelo, porque aunque ambos dependen de  $A_{11}$ , no dependen el uno del otro. Finalmente, podemos completar  $A_{22}$  recursivamente.

- Proporcione un pseudocódigo de subprocessos múltiples que realice este cálculo de plantilla simple utilizando un algoritmo de divide y vencerás SIMPLE-STENCIL basado en la posición de descomposición (27.11) y la discusión anterior. (No se preocupe por los detalles del caso base, que depende de la plantilla específica). Proporcione y resuelva recurrencias para el trabajo y el intervalo de este algoritmo en términos de  $n$ . ¿Qué es el paralelismo?
- Modifique su solución a la parte (a) para dividir una matriz  $nn$  en nueve subarreglos  $n=3$   $n=3$ , recurriendo nuevamente con tanto paralelismo como sea posible. Analiza este algoritmo. ¿Cuánto más o menos paralelismo tiene este algoritmo en comparación con el algoritmo de la parte (a)?
- Generalice sus soluciones a las partes (a) y (b) de la siguiente manera. Elija un entero  $b \geq 2$ . Divida una matriz  $nn$  en subarreglos  $b^2$ , cada uno de tamaño  $n=b^2$ , recursivos con tanto paralelismo como sea posible. En términos de  $n$  y  $b$ , ¿cuáles son el trabajo, el intervalo y el paralelismo de su algoritmo? Argumente que, utilizando este enfoque, el paralelismo debe ser  $\frac{1}{b^2}$  para cualquier elección de  $b \geq 2$ . (Sugerencia: para este último argumento, demuestre que el exponente de  $n$  en el paralelismo es estrictamente menor que 1 para cualquier elección de  $b \geq 2$ ).

d. Proporcione un pseudocódigo para un algoritmo de subprocessos múltiples para este cálculo de plantilla simple que logra un paralelismo  $\sim n = \lg n$ . Argumente usando nociones de trabajo y lapso que el problema, de hecho, tiene un paralelismo  $\sim n$  inherente. Resulta que la naturaleza de divide y vencerás de nuestro pseudocódigo multiproceso no nos permite lograr este paralelismo máximo.

#### 27-6 Algoritmos aleatorios de subprocessos múltiples AI

igual que con los algoritmos en serie ordinarios, a veces queremos implementar algoritmos aleatorios de subprocessos múltiples. Este problema explora cómo adaptar las diversas medidas de rendimiento para manejar el comportamiento esperado de dichos algoritmos.

También le pide que diseñe y analice un algoritmo de subprocessos múltiples para una ordenación rápida aleatoria.

a. Explique cómo modificar la ley del trabajo (27.2), la ley del intervalo (27.3) y el límite del programador codicioso (27.4) para trabajar con expectativas cuando  $TP$ ,  $T1$  y  $T2$  son todas variables aleatorias.

b. Considere un algoritmo aleatorio de subprocessos múltiples para el cual el 1% del tiempo tenemos  $T1 D 104$  y  $T10;000 D 1$ , pero el 99% del tiempo tenemos  $T1 D T10;000 D 109$ . Argumente que la aceleración de un algoritmo aleatorio de subprocessos múltiples el ritmo debe definirse como  $E CET1 = E CETP$ , en lugar de  $E CET1=TP$ .

C. Argumente que el paralelismo de un algoritmo aleatorio de subprocessos múltiples debe definirse como la relación  $E CET1 = E CET1$ .

d. Multiprocesamiento del algoritmo RANDOMIZED-QUICKSORT en la página 179 mediante el uso de paralelismo anidado. (No paralelice RANDOMIZED-PARTITION). Proporcione el pseudocódigo para su algoritmo P-RANDOMIZED-QUICKSORT .

mi. Analice su algoritmo de subprocessos múltiples para la ordenación rápida aleatoria. (Sugerencia: revise el análisis de SELECCIÓN ALEATORIA en la página 216).

#### Notas del capítulo

Las computadoras paralelas, los modelos para computadoras paralelas y los modelos algorítmicos para la programación paralela han existido en varias formas durante años. Las ediciones anteriores de este libro incluían material sobre redes de clasificación y el modelo PRAM (Máquina de acceso aleatorio paralelo). El modelo de datos paralelos [48, 168] es otro modelo popular de programación algorítmica, que presenta operaciones en vectores y matrices como primitivas.

Graham [149] y Brent [55] demostraron que existen planificadores que alcanzan el límite del Teorema 27.1. Eager, Zahorjan y Lazowska [98] demostraron que cualquier programador ambicioso logra este límite y propusieron la metodología de usar trabajo y intervalo (aunque no por esos nombres) para analizar algoritmos paralelos. Blelloch [47] desarrolló un modelo de programación algorítmica basado en el trabajo y el intervalo (al que llamó la "profundidad" del cálculo) para la programación de datos en paralelo. Blumofe y Leiserson [52] dieron un algoritmo de programación distribuida para subprocessos múltiples dinámicos basado en "robo de trabajo" aleatorio y demostraron que logra el límite  $E \leq T_1 = PC O.T_1/$ . Arora, Blumofe y Plaxton [19] y Blelloch, Gibbons y Matias [49] también proporcionaron algoritmos demostrablemente buenos para programar cálculos dinámicos de subprocessos múltiples.

El pseudocódigo de subprocessos múltiples y el modelo de programación estuvieron fuertemente influenciados por el proyecto Cilk [51, 118] en el MIT y las extensiones Cilk++ [71] para C++ distribuidas por Cilk Arts, Inc. Muchos de los algoritmos de subprocessos múltiples en este capítulo aparecieron en notas de conferencias no publicadas, por CE Leiserson y H. Prokop y han sido implementados en Cilk o Cilk++. El algoritmo de clasificación por combinación de subprocessos múltiples se inspiró en un algoritmo de Akl [12].

La noción de consistencia secuencial se debe a Lamport [223].

---

## 28

## Operaciones Matriciales

Debido a que las operaciones con matrices se encuentran en el corazón de la computación científica, los algoritmos eficientes para trabajar con matrices tienen muchas aplicaciones prácticas. Este capítulo se enfoca en cómo multiplicar matrices y resolver conjuntos de ecuaciones lineales simultáneas. El Apéndice D repasa los conceptos básicos de las matrices.

La sección 28.1 muestra cómo resolver un conjunto de ecuaciones lineales usando descomposiciones LUP. Luego, la Sección 28.2 explora la estrecha relación entre la multiplicación y la inversión de matrices. Finalmente, la Sección 28.3 analiza la importante clase de matrices definidas positivas simétricas y muestra cómo podemos usarlas para encontrar una solución de mínimos cuadrados para un conjunto sobre determinado de ecuaciones lineales.

Una cuestión importante que surge en la práctica es la estabilidad numérica. Debido a la precisión limitada de las representaciones de coma flotante en las computadoras reales, los errores de redondeo en los cálculos numéricos pueden amplificarse en el transcurso de un cálculo, lo que lleva a resultados incorrectos; llamamos a tales cálculos numéricamente inestables. Aunque de vez en cuando consideraremos brevemente la estabilidad numérica, no nos centraremos en ella en este capítulo. Lo remitimos al excelente libro de Golub y Van Loan [144] para una discusión detallada de los problemas de estabilidad.

---

### 28.1 Resolución de sistemas de ecuaciones lineales

Numerosas aplicaciones necesitan resolver conjuntos de ecuaciones lineales simultáneas. Podemos formular un sistema lineal como una ecuación matricial en la que cada elemento de la matriz o del vector pertenece a un campo, normalmente los números reales  $\mathbb{R}$ . Esta sección explica cómo resolver un sistema de ecuaciones lineales mediante un método llamado descomposición LUP.

Empezamos con un conjunto de ecuaciones lineales en  $n$  incógnitas  $x_1; x_2; \dots; x_n$ :

$$a_{11}x_1 + a_{12}x_2 = b_1; \quad a_{21}x_1 + a_{22}x_2 = b_2; \\ \vdots \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 = b_n;$$

(28.1)

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1; \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2; \\ \vdots \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n;$$

Una solución a las ecuaciones (28.1) es un conjunto de valores para  $x_1, x_2, \dots, x_n$  que satisfacen todas las ecuaciones simultáneamente. En esta sección, tratamos solo el caso en el que hay exactamente  $n$  ecuaciones en  $n$  incógnitas.

Podemos reescribir convenientemente las ecuaciones (28.1) como la ecuación matriz-vector

$$\begin{matrix} & a_{1n} \\ a_{21} & a_{22} & & a_{2n} & x_2 & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & x_1 & x_n \\ & & & & & b_1 \end{matrix}$$

o, de manera equivalente, dejando  $A$  para la matriz de coeficientes,  $x$  para el vector de variables y  $b$  para el vector de términos independientes,

$$Ax = b \quad (28.2)$$

Si  $A$  no es singular, posee un inverso  $A^{-1}$ , y

$$x = A^{-1}b \quad (28.3)$$

es el vector solución. Podemos demostrar que  $x$  es la solución única de la ecuación (28.2) de la siguiente manera. Si hay dos soluciones,  $x$  y  $x_0$  entonces  $Ax = Ax_0 = b$ , dejando que  $I$  denote una matriz identidad,

$$x = Ix$$

$$= I(A^{-1}b)$$

$$= A^{-1}(Ab)$$

$$= x_0$$

En esta sección, nos ocuparemos predominantemente del caso en el que  $A$  no es singular o, de manera equivalente (por el teorema D.1), el rango de  $A$  es igual al número  $n$  de incógnitas. Hay otras posibilidades, sin embargo, que merecen una breve discusión. Si el número de ecuaciones es menor que el número  $n$  de incógnitas o, más generalmente, si el rango de  $A$  es menor que  $n$ , entonces el sistema está indeterminado. Un sistema indeterminado normalmente tiene infinitas soluciones, aunque puede no tener ninguna solución si las ecuaciones son inconsistentes. Si el número de ecuaciones excede el número  $n$  de incógnitas, el sistema está sobre determinado y es posible que no exista ninguna solución. La sección 28.3 aborda la importante

problema de encontrar buenas soluciones aproximadas a sistemas sobredeterminados de ecuaciones lineales.

Volvamos a nuestro problema de resolver el sistema  $Ax = b$  de  $n$  ecuaciones en  $n$  incógnitas. Podríamos calcular  $A^{-1}$  y luego, usando la ecuación (28.3), multiplicar  $b$  por  $A^{-1}$ , obteniendo  $x = A^{-1}b$ . Este enfoque adolece en la práctica de inestabilidad numérica. Afortunadamente, otro enfoque, la descomposición LUP, es numéricamente estable y tiene la ventaja adicional de ser más rápido en la práctica.

#### Descripción general de la descomposición LUP

La idea detrás de la descomposición LUP es encontrar tres matrices  $P, L$  y  $U$  tales que

$$PAx = LUx \quad (28.4)$$

dónde

$L$  es una matriz triangular inferior unitaria,

$U$  es una matriz triangular superior y  $P$  es

una matriz de permutación.

Llamamos a las matrices  $L$ ,  $U$  y  $P$  que satisfacen la ecuación (28.4) una descomposición LUP de la matriz  $A$ . Mostraremos que toda matriz  $A$  no singular posee tal descomposición.

Calcular una descomposición LUP para la matriz  $A$  tiene la ventaja de que podemos resolver más fácilmente los sistemas lineales cuando son triangulares, como es el caso de las matrices  $L$  y  $U$ . Una vez que hemos encontrado una descomposición LUP para  $A$ , podemos resolver la ecuación (28.2),  $Ax = b$ , resolviendo solo sistemas lineales triangulares, como sigue. Al multiplicar ambos lados de  $Ax = b$  por  $P$  se obtiene la ecuación equivalente  $PAx = Pb$ , que, según el ejercicio D.1-4, equivale a permutar las ecuaciones (28.1).

Usando nuestra descomposición (28.4), obtenemos

$$LUx = Pb$$

Ahora podemos resolver esta ecuación resolviendo dos sistemas lineales triangulares. Definamos  $y = Dx$ , donde  $x$  es el vector solución deseado. Primero, resolvemos el sistema triangular inferior

$$Ly = Pb \quad (28.5)$$

para el vector desconocido  $y$  por un método llamado "sustitución directa". Habiendo resuelto para  $y$ , luego resolvemos el sistema triangular superior

$$Ux = y \quad (28.6)$$

para la  $x$  desconocida por un método llamado "sustitución hacia atrás". Como la matriz de permutación  $P$  es invertible (ejercicio D.2-3), al multiplicar ambos lados de equa se que ción (28.4) por  $P^{-1}$  obtiene  $P^{-1}PA = P^{-1}LU$ , de modo

$$ADP^{-1}LUx = b \quad (28.7)$$

Por lo tanto, el vector  $x$  es nuestra solución para  $Ax = b$ :

$$Ax = LUx \quad (\text{por la ecuación (28.7)})$$

$$Dx = Ly \quad (\text{por la ecuación (28.6)})$$

$$Dx = Pb \quad (\text{por la ecuación (28.5)})$$

$$x = b$$

Nuestro siguiente paso es mostrar cómo funciona la sustitución hacia adelante y hacia atrás y luego atacar el problema de calcular la descomposición LUP en sí.

#### Sustitución hacia adelante y hacia atrás

La sustitución directa puede resolver el sistema triangular inferior (28.5) en  $n^2$  tiempo, dados  $L$ ,  $P$  y  $b$ . Por conveniencia, representamos la permutación  $P$  de forma compacta mediante un arreglo  $\{1 : n\}$ . Para  $i = 1; 2; \dots; n$ , la entrada  $\{i\}$  indica que  $P_i = i$  y  $P_{ij} = 0$  para  $j \neq i$ . Así,  $PA$  tiene  $a_{ij}$  en la fila  $i$  y columna  $j$ , y  $Pb$  tiene  $b_{i}$  como su  $i$ -ésimo elemento. Como  $L$  es un triángulo inferior unitario, podemos reescribir la ecuación (28.5) como

$$\begin{array}{ll} re b & \{1\}; \\ y_1 = b & \{2\}; \\ l_{11}y_1 + b & \{3\}; \\ l_{21}y_1 + b & \{4\}; \\ l_{31}y_1 + b & \{5\}; \\ \vdots & \vdots \end{array}$$

$$l_{n1}y_1 + b = b \quad \{n\};$$

La primera ecuación nos dice que  $y_1 = b / \{1\}$ . Conociendo el valor de  $y_1$ , podemos sustituirla en la segunda ecuación, obteniendo

$$y_2 = b - l_{21}y_1 \quad \{2\};$$

Ahora, podemos sustituir tanto  $y_1$  como  $y_2$  en la tercera ecuación, obteniendo

$$y_3 = b - l_{31}y_1 - l_{32}y_2 \quad \{3\};$$

En general, sustituimos  $y_1; y_2; \dots; y_i$  "adelante" en la  $i$ -ésima ecuación para resolver  $y_i$ :

i1

$$y_i \leftarrow b\sigma(x_i) + y_j :_{j \neq i}$$

Habiendo resuelto para  $y$ , resolvemos para  $x$  en la ecuación (28.6) usando sustitución hacia atrás, que es similar a la sustitución hacia adelante. Aquí, primero resolvemos la  $n$ -ésima ecuación y trabajamos hacia atrás hasta la primera ecuación. Al igual que la sustitución hacia adelante, este proceso se ejecuta en tiempo  $\dots n^2$ . Como  $U$  es triangular superior, podemos reescribir el sistema (28.6) como

un2;n2xn2 C un2;n1xn1 C un2;nxn D yn2 ;

un1;n1xn1 C un1;nxn D yn1

un;nxn D yn

Por lo tanto, podemos resolver para  $x_n; x_{n-1}; \dots; x_1$  sucesivamente como sigue:

$x_n \leftarrow y_n = u_n; n ; x_{n+1} \leftarrow .y_{n+1}$

$\text{un1}; \text{nxn} = \text{un1}; \text{n1} ; \text{xn2} \text{ D .yn2 .un2;n1xn1 C un2;nxn/}$   
 $= \text{un2;n2} ;$

o, en general.

$x_i \neq y_i \forall i$

Dados  $P$ ,  $L$ ,  $U$  y  $b$ , el procedimiento LUP-SOLVE resuelve  $x$  combinando la sustitución hacia adelante y hacia atrás. El pseudocódigo asume que la dimensión  $n$  aparece en el atributo  $L:\text{filas}$  y que la matriz de permutación  $P$  está representada por la matriz:

| LUP-BESOI VERB | : tur : b/ 1 n D

| :filas 2 sea x un nuevo

vector de longitud n-3 para i D 1 a n

4 yi D bŒi Pi1 lij yj  
iD1 5 para i D n

yi Pn 6 hasta 1 xix D=ui i

7 volver x

El procedimiento LUP-SOLVE resuelve y usando sustitución hacia adelante en las líneas 3 y 4, y luego resuelve x usando sustitución hacia atrás en las líneas 5 y 6. Dado que la suma dentro de cada uno de los ciclos for incluye un ciclo implícito, el tiempo de ejecución es ,n2/.

Como ejemplo de estos métodos, considere el sistema de ecuaciones lineales definido por

$$\begin{array}{ccc} 120 & & 3 \\ 344 & xD & 7 \\ 563 & & 8 \end{array}$$

dónde

$$\begin{array}{ccc} & 120 & \\ \text{ANUNCIO} & 344 & : \\ & 563 & \\ & 3 & \\ b D & 7 & : \\ & 8 & \end{array}$$

y deseamos resolver para la incógnita x. La descomposición LUP es

$$\begin{array}{ccc} & 1 00 & \\ \text{LD} & 0:2 & 1 0 \\ & 0:6 0:5 1 & : \\ & 56 3 & \\ \text{UD} & 0 0:8 & 0:6 \\ & 0 0 2:5 & : \\ & 001 & \\ \text{PD} & 100 & \\ & 010 & \end{array}$$

(Es posible que desee verificar que PA D LU). Usando la sustitución hacia adelante, resolvemos Ly DP b para y:

$$\begin{array}{ccc} 1 00 0:2 1 0 & y1 & 8 \\ 0:6 0:5 1 & y2 & 3 \\ & y3 & 7 \end{array}$$

obtención

$$\begin{array}{ccc} y D & 8 \\ & 1:4 1:5 \end{array}$$

calculando primero y1, luego y2 y finalmente y3. Usando sustitución hacia atrás, resolvemos Ux D y para x:

56 3 0 0:8 0:6 0 0      x1  
2:5                        x2      D      8  
                              x3      1:4 1:5

obteniendo así la respuesta deseada

1:4  
2:2  
0:6

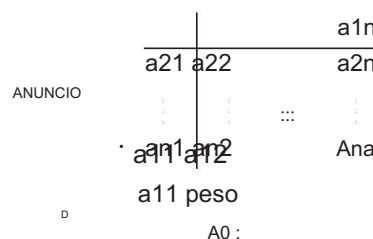
calculando primero x3, luego x2 y finalmente x1

Cálculo de una descomposición LU

Ahora hemos demostrado que si podemos crear una descomposición LUP para una matriz A no singular, entonces la sustitución hacia adelante y hacia atrás puede resolver el sistema Ax = b de ecuaciones lineales. Ahora mostramos cómo calcular eficientemente una descomposición LUP para A. Comenzamos con el caso en el que A es una matriz no singular y P está ausente (o, de manera equivalente, PD In). En este caso, factorizamos AD LU. Llamamos a las dos matrices L y U una descomposición LU de A.

Usamos un proceso conocido como eliminación gaussiana para crear una descomposición LU. Comenzamos restando múltiplos de la primera ecuación de las otras ecuaciones para eliminar la primera variable de esas ecuaciones. Luego, restamos múltiplos de la segunda ecuación de la tercera y las siguientes ecuaciones, de modo que ahora se eliminan la primera y la segunda variable. Continuamos este proceso hasta que el sistema que queda tiene una forma triangular superior; de hecho, es la matriz U. La matriz L está formada por los multiplicadores de fila que hacen que se eliminen las variables.

Nuestro algoritmo para implementar esta estrategia es recursivo. Deseamos construir una descomposición LU para una matriz no singular  $A_{nn}$ . Si  $n = 1$ , entonces hemos terminado, ya que podemos elegir  $L = I_1$  y  $U = A$ . Para  $n > 1$ , dividimos  $A$  en cuatro partes:



donde es un vector de columna .n 1/, wT es un vector de fila .n 1/ y A0 es una matriz de .n 1/ .n 1/. Luego, usando álgebra matricial (verifique las ecuaciones por

simplemente multiplicando), podemos factorizar A como

$$\begin{array}{llll}
 & a11 \text{ peso} & & \\
 \text{ANUNCIO} & A0 & & \\
 & & & \\
 & 1 0 = a11 & a11 & wT \\
 & en 1 & 0 A0 & wT=a11: \\
 \text{D} & & & 
 \end{array} \tag{28.8}$$

Los ceros en la primera y segunda matriz de la ecuación (28.8) son vectores fila y columna .n 1/, respectivamente. El término  $w^T = a_{11}$ , formado tomando el producto exterior de y w y dividiendo cada elemento del resultado por  $a_{11}$ , es una matriz .n 1/ .n 1/, que se ajusta en tamaño a la matriz A0 de la que se resta . La resultante .n 1/ .n 1/ matriz A0  $w^T = a_{11}$

(28.9)

se llama complemento de Schur de A con respecto a a11.

Decimos que si  $A$  es no singular, entonces el complemento de Schur también es no singular. ¿Por qué? Supongamos que el complemento de Schur, que es  $.n \cdot 1 / .n \cdot 1$ , es singular. Entonces, por el teorema D.1, tiene un rango de fila estrictamente menor que  $n - 1$ . Debido a que las  $n - 1$  entradas inferiores en la primera columna de la matriz

a11 pesc  
0 A0 wT=a11

son todas 0, las  $n-1$  filas inferiores de esta matriz deben tener un rango de fila estrictamente menor que  $n-1$ . Por lo tanto, el rango de fila de toda la matriz es estrictamente menor que  $n$ . Aplicando el ejercicio D.2-8 a la ecuación (28.8),  $A$  tiene un rango estrictamente menor que  $n$ , y del teorema D.1 derivamos la contradicción de que  $A$  es singular.

Debido a que el complemento de Schur no es singular, ahora podemos encontrar recursivamente una Descomposición LU para ello. Digamos que

A0 wT=a11 D10 U0

donde  $L_0$  es unidad triangular inferior y  $U_0$  es triangular superior. Entonces, usando álgebra matricial, tenemos

	1	a11	wT
ANUNCIO	0 =a11 en 1	0 A0 wT=a11 a11 wT	
	1		
D	0 =a11 en 1	0 L0 U0	
	0	a11 peso	
D	1 =a11 L0	0 U0	
DLU	:		

proporcionando así nuestra descomposición LU. (Tenga en cuenta que debido a que  $L_0$  es triangular inferior unitario, también lo es L, y debido a que  $U_0$  es triangular superior, también lo es U).

Por supuesto, si  $a_{11} = 0$ , este método no funciona, porque divide por 0. Tampoco  $wT = a_{11}$  si la entrada superior izquierda del complemento de Schur  $A_{00}$  es 0, ya que \_\_\_\_\_ funciona dividimos por ella en el siguiente paso de la recursividad. Los elementos por los que dividimos durante la descomposición LU se denominan pivotes y ocupan los elementos diagonales de la matriz U. La razón por la que incluimos una matriz de permutación P durante la descomposición LUP es que nos permite evitar dividir por 0. Cuando usamos permutaciones para evitar la división por 0 (o por números pequeños, lo que contribuiría a la inestabilidad numérica), estamos pivoteando.

Una clase importante de matrices para las que la descomposición LU siempre funciona correctamente es la clase de matrices definidas positivas simétricas. Tales matrices no requieren pivotamiento y, por lo tanto, podemos emplear la estrategia recursiva descrita anteriormente sin temor a dividir por 0. Probaremos este resultado, así como varios otros, en la Sección 28.3.

Nuestro código para la descomposición LU de una matriz A sigue la estrategia recursiva, excepto que un bucle de iteración reemplaza la recursividad. (Esta transformación es una optimización estándar para un procedimiento "recursivo de cola", uno cuya última operación es una llamada recursiva a sí mismo. Vea el problema 7-4.) Se supone que el atributo A: rows da la dimensión de A. inicialice la matriz U con 0 debajo de la diagonal y la matriz L con 1 en su diagonal y 0 arriba de la diagonal.

#### LU-DESCOMPOSICIÓN.A/

1 n D A: filas

2 sean L y U nuevas matrices nn

3 inicialice U con 0s debajo de la diagonal 4

inicialice L con 1s en la diagonal y 0s arriba de la diagonal 5 para k D 1 a n 6

ukk D akk

7 para i D k C 1 to n lik

D aik=ukk // lik i

uki D aki aguanta // uki aguanta wT

8 para i D k C 1 to n

9 para j D k C 1 to n aij

10 D aij likukj

11 12 13 volver L y U

El ciclo for externo que comienza en la línea 5 itera una vez por cada paso recursivo. Dentro de este bucle, la línea 6 determina que el pivote sea  $ukk = D_{akk}$ . El ciclo for en las líneas 7–9 (que no se ejecuta cuando  $k > n$ ), usa los vectores y  $wT$  para actualizar L en lik, y la línea 9 y determina los elementos del vector, almacenando i calcula los elementos U. La línea 8 del vector  $wT$ , almacenando  $wT$  en  $uki$ . Finalmente, las líneas 10 a 12 calculan los elementos del complemento de Schur y los almacenan nuevamente en el ma-

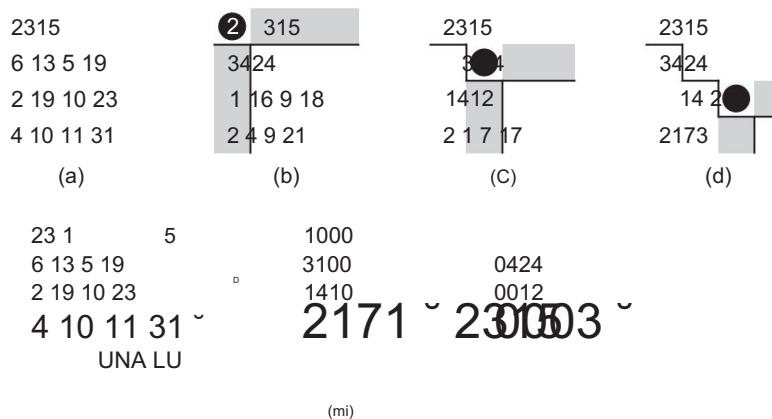


Figura 28.1 La operación de LU-DECOMPOSICIÓN. (a) La matriz A. (b) El elemento  $a_{11}$  D 2 en el círculo negro es el pivote, la columna sombreada es  $=a_{11}$  y la fila sombreada es  $wT$ . Los elementos de U calculados hasta ahora están por encima de la línea horizontal, y los elementos de L están a la izquierda de la  $wT=a_{11}$  ocupa la parte inferior matriz de complemento de Schur  $A_0$  opera sobre la derecha. (c) Ahora trazamos una línea vertical. La matriz de complemento de Schur producida a partir de la parte (b). El elemento  $a_{22}$  D 4 en el círculo negro es el pivote, y la columna y la fila sombreadas son  $=a_{22}$  y  $wT$  (en la partición del complemento de Schur), respectivamente. Las líneas dividen la matriz en los elementos de U calculados hasta ahora (arriba), los elementos de L calculados hasta ahora (izquierda) y el nuevo complemento de Schur (abajo a la derecha). (d) Después del siguiente paso, se factoriza la matriz A. (El elemento 3 en el nuevo complemento de Schur se convierte en parte de U cuando termina la recursividad.) (e) La factorización AD LU.

matriz A. (No necesitamos dividir por  $akk$  en la línea 12 porque ya lo hicimos cuando calculamos  $lik$  en la línea 8). Debido a que la línea 12 está triplemente anidada, LU-DECOMPOSITION se ejecuta en el tiempo  $,n^3/$ .

La Figura 28.1 ilustra la operación de DESCOMPOSICIÓN LU. Muestra una optimización estándar del procedimiento en el que almacenamos los elementos significativos de L y U en la matriz A. Es decir, podemos establecer una correspondencia entre cada elemento  $a_{ij}$  y  $l_{ij}$  (si  $i > j$ ) o  $u_{ij}$  (si  $i < j$ ) y actualice la matriz A para que contenga tanto L como U cuando finalice el procedimiento. Para obtener el pseudocódigo para esta optimización del pseudocódigo anterior, simplemente reemplace cada referencia a l o u por a; puedes verificar fácilmente que esta transformación conserva exactitud.

#### Cálculo de una descomposición LUP

Generalmente, al resolver un sistema de ecuaciones lineales  $Ax = b$ , debemos pivotar sobre los elementos diagonales de A para evitar dividir por 0. Dividir por 0 sería, por supuesto, desastroso. Pero también queremos evitar dividir por un valor pequeño, incluso si A es

nonsingular—porque pueden resultar inestabilidades numéricas. Por lo tanto, tratamos de pivotar sobre un valor grande.

Las matemáticas detrás de la descomposición LUP son similares a las de la descomposición LU. Recuerde que se nos da una matriz A no singular de  $n \times n$ , y deseamos encontrar una matriz de permutación P, una matriz triangular inferior unitaria L y una matriz triangular superior U tal que  $PA = LU$ . Antes de dividir la matriz A, como hicimos con la descomposición LU, movemos un elemento distinto de cero, digamos  $a_{k1}$ , de algún lugar de la primera columna a la posición 1;1 de la matriz. Para la estabilidad numérica, elegimos  $a_{k1}$  como el elemento de la primera columna con el mayor valor absoluto. (La primera columna no puede contener solo 0, porque entonces A sería singular, porque su determinante sería 0, por los teoremas D.4 y D.5.) Para preservar el conjunto de ecuaciones, intercambiamos el renglón 1 con el renglón k, que es equivalente a multiplicar A por una matriz de permutación Q a la izquierda (Ejercicio D.1-4). Por lo tanto, podemos escribir:

$$\begin{array}{c} \text{ak1 peso} \\ \text{control de calidad} \\ A_0 : \end{array}$$

donde  $D = a_{21}; a_{31}; \dots; a_{n1}/T$ , excepto que  $a_{11}$  reemplaza a  $a_{k1}$ ;  $w^T = a_{k2}; a_{k3}; \dots; a_{kn}/$ ; y  $A_0$  es una matriz  $(n-1) \times (n-1)$ . Dado que  $a_{k1} \neq 0$ , ahora podemos realizar el mismo álgebra lineal que para la descomposición LU, pero ahora garantizamos que no dividiremos por 0:

$$\begin{array}{c} \text{ak1 peso} \\ \text{control de calidad} \\ A_0 \\ \\ \begin{matrix} & 1 & & \text{ak1} & & \text{peso} \\ D & 0 = a_{k1} \text{ en } 1 & & 0 & A_0 & w^T = a_{k1} : \end{matrix} \end{array}$$

Como vimos para la descomposición LU, si A es no singular, entonces el complemento de Schur  $A_0 w^T = a_{k1}$  también es no singular. Por lo tanto, podemos encontrar recursivamente una descomposición LUP para él, con matriz triangular inferior unitaria  $L_0$ , matriz triangular superior  $U_0$ , y matriz de permutación  $P_0$ , tal que

$$P_0 A_0 w^T = a_{k1} / D L_0 U_0 :$$

Definir

$$PD = \begin{pmatrix} 1 & 0 \\ 0 & P_0 \end{pmatrix} q;$$

que es una matriz de permutación, ya que es el producto de dos matrices de permutación (Ejercicio D.1-4). ahora tenemos

```

PA D      1 0
          0 P0   control de calidad

      1 0      1 0 =ak1      ak1
D     0 P0      en 1           wT 0 A0 wT=ak1

      1 0      ak1
D     P0 =ak1 P0      wT 0 A0 wT=ak1

      1      0      ak1
D     P0 =ak1 en 1      wT 0 P0 .A0 wT=ak1/

      1      0      ak1 wT 0
D     P0 =ak1 en 1      L0 U0

      1 0      ak1 peso
D     P0 =ak1 L0      0U0 _

DLU      :

```

dando la descomposición LUP. Como  $L_0$  es triangular inferior unitario, también lo es  $L$ , y como  $U_0$  es triangular superior, también lo es  $U$ .

Observe que en esta derivación, a diferencia de la descomposición LU, debemos  $wT=ak1$  multiplicar tanto el vector columna  $=ak1$  como el complemento de Schur  $A0$  por la matriz de permutación  $P0$ . Aquí está el pseudocódigo para la descomposición LUP:

#### LUP-DESCOMPOSICIÓN.A/

```

1 n D A:filas 2 sea
OE1 : : n un nuevo arreglo 3 para i
D 1 a n OEi D i 4 5
    para k
D 1 a n 6 p D 0 para
i D k a n si jaikj >
    pp D jaikj k0 D i
8        si p == 0
9            error
10           "matriz
11           singular"
12           intercambiar OEk con OEk0
13           por i D 1 a n intercambiar
14           aki con ak0i por
15           i D k C 1 a n aik D aik=akk
16           por j D k C 1 a n aij D
17           aij aikakj
18
19

```

Al igual que LU-DECOMPOSITION, nuestro procedimiento LUP-DECOMPOSITION reemplaza la recursión con un ciclo de iteración. Como una mejora sobre la implementación directa de la recursividad, mantenemos dinámicamente la matriz de permutación P como matriz , un donde  $\text{C}Ei D j$  significa que la i-ésima fila de P contiene un 1 en la columna j. también implementamos el código para calcular L y U "en su lugar" en la matriz A. Por lo tanto, cuando finaliza el procedimiento,

si  $i > j$ ;

$a_{ij} D ( l_{ij} u_{ij} \text{ si } ij )$  :

La figura 28.2 ilustra cómo la DESCOMPOSICIÓN LUP factoriza una matriz. Las líneas 3 y 4 inicializan la matriz para representar la permutación de identidad. El bucle for exterior que comienza en la línea 5 implementa la recursividad. Cada vez que pasan por el ciclo externo, las líneas 6 a 10 determinan el elemento  $a_{k0k}$  con el valor absoluto más grande de los de la primera columna actual (columna k) de la matriz .nk C 1/ .nk C 1/ cuya descomposición LUP estamos encontrando. Si todos los elementos de la primera columna actual son cero, las líneas 11 y 12 informan que la matriz es singular. Para pivotar, intercambiamos  $\text{C}Ek0$  con  $\text{C}Ek$  en la línea 13 e intercambiamos las filas kth y k0th de A en las líneas 14–15, lo que hace que el elemento pivote sea  $a_{kk}$ . (Todas las filas se intercambian porque en la derivación del método anterior, no solo se multiplica  $A_0 wT = a_{k1}$  por  $P_0$  sino también  $= a_{k1}$ ). Finalmente, el complemento de Schur se calcula mediante las líneas 16 a 19 de la misma manera que se calcula mediante las líneas 7 a 12 de LU-DECOMPOSITION, excepto que aquí la operación está escrita para funcionar en su lugar.

Debido a su estructura de bucle anidado triple, LUP-DECOMPOSITION tiene un tiempo de ejecución de „n<sup>3</sup>”, que es el mismo que el de LU-DECOMPOSITION. Así, pivotar nos cuesta a lo sumo un factor constante en el tiempo.

## Ejercicios

### 28.1-1

Resuelva la ecuación

$$\begin{array}{ccc|c} 100 & 410 & & x_1 \\ & x_2 & \dots & 3 \\ 651 & & x_3 & 7 \end{array}$$

mediante el uso de sustitución hacia adelante.

### 28.1-2

Encuentre una descomposición LU de la matriz

$$\begin{matrix} 4 & 5 & 6 \\ 8 & 6 & 7 \\ 12 & 7 & 12 \end{matrix}$$

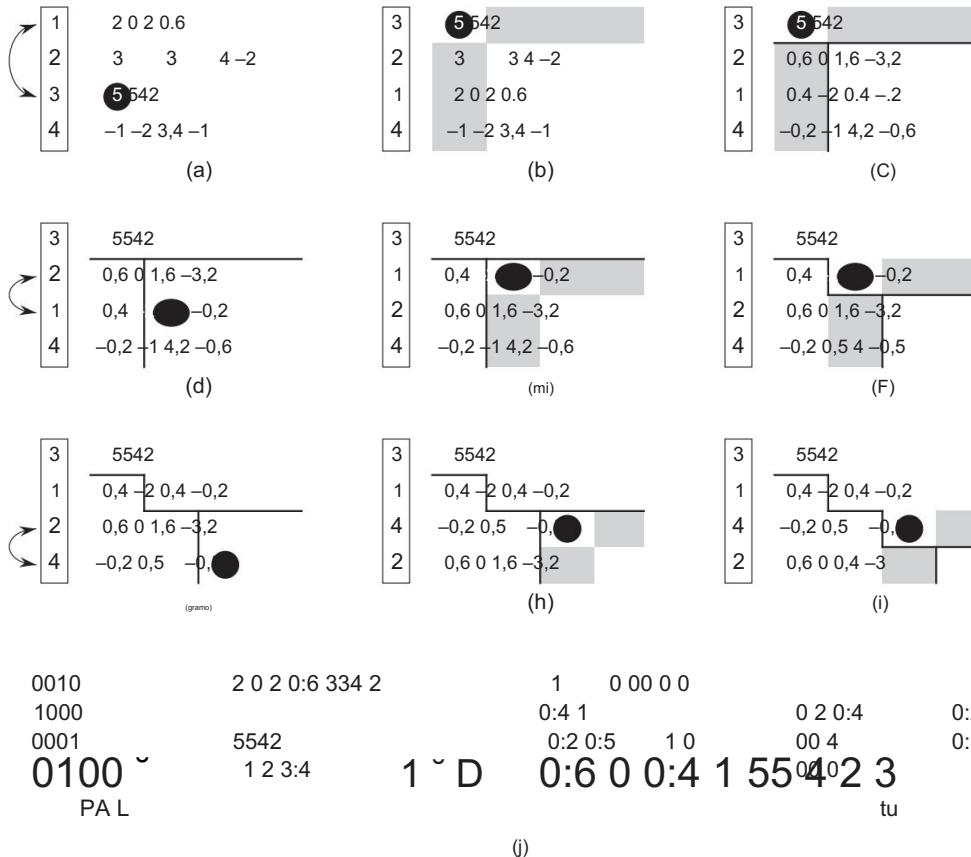


Figura 28.2 La operación de LUP-DECOMPOSITION. (a) La matriz de entrada A con la permutación de identidad de las filas de la izquierda. El primer paso del algoritmo determina que el elemento 5 en el círculo negro en la tercera fila es el pivote de la primera columna. (b) Las filas 1 y 3 se intercambian y la permutación se actualiza. La columna y la fila sombreadas representan y wT. (c) El vector se reemplaza por =5 y la parte inferior derecha de la matriz se actualiza con el complemento de Schur. Las líneas dividen la matriz en tres regiones: elementos de U (arriba), elementos de L (izquierda) y elementos del complemento de Schur (abajo a la derecha). (d)–(f) El segundo paso. (g)–(i) El tercer paso. No se producen más cambios en el cuarto (último) paso. (j) La descomposición LUP PA D LU.

28.1-3

Resuelva la ecuación

$$\begin{array}{ccc|c} 154 & x_1 & & 12 \\ 203 & x_2 & & 9 \\ 582 & & & \\ \hline & x_3 & & 5 \end{array}$$

utilizando una descomposición LUP.

28.1-4

Describir la descomposición LUP de una matriz diagonal.

28.1-5

Describa la descomposición LUP de una matriz de permutación A y demuestre que es única.

28.1-6

Muestre que para todo  $n \geq 1$ , existe una matriz  $n \times n$  singular que tiene una posición de descomposición LU.

28.1-7

En LU-DECOMPOSITION, ¿es necesario realizar la iteración del bucle for más exterior cuando  $k < D - n$ ? ¿Qué tal en LUP-DESCOMPOSICIÓN?

## 28.2 Matrices inversoras

Aunque en la práctica generalmente no usamos matrices inversas para resolver sistemas de ecuaciones lineales, prefiriendo usar técnicas más estables numéricamente como la descomposición LUP, a veces necesitamos calcular una matriz inversa. En esta sección, mostramos cómo usar la descomposición LUP para calcular una matriz inversa.

También demostramos que la multiplicación de matrices y el cálculo de la inversa de una matriz son problemas equivalentemente difíciles, en el sentido de que (sujeto a condiciones técnicas) podemos usar un algoritmo para que uno resuelva el otro en el mismo tiempo de ejecución asintótico.

Por lo tanto, podemos usar el algoritmo de Strassen (ver Sección 4.2) para la multiplicación de matrices para invertir una matriz. De hecho, el artículo original de Strassen estaba motivado por el problema de mostrar que un conjunto de ecuaciones lineales podía resolverse más rápidamente que con el método habitual.

### Cálculo de una matriz inversa a partir de una descomposición LUP

Supongamos que tenemos una descomposición LUP de una matriz A en forma de tres matrices L, U y P tales que  $PA = LU$ . Usando LUP-SOLVE, podemos resolver una ecuación de la forma  $Ax = b$  en el tiempo  $,n^2/$ . Dado que la descomposición LUP depende de A pero no de b, podemos ejecutar LUP-SOLVE en un segundo conjunto de ecuaciones de la forma  $Ax = b_0$  en un tiempo adicional  $,n^2/$ . En general, una vez que tenemos la descomposición LUP de A, podemos resolver, en el tiempo  $,kn^2/$ , k versiones de la ecuación  $Ax = b$  que difieren solo en b.

Podemos pensar en la ecuación

$$AX = D \quad \text{Entrada ;} \quad (28.10)$$

que define la matriz X, la inversa de A, como un conjunto de n ecuaciones distintas de la forma  $Ax = b$ . Para ser precisos, denotemos con  $X_i$  la i-ésima columna de X, y recordemos que el vector unitario  $e_i$  es la i-ésima columna de  $I_n$ . Entonces podemos resolver la ecuación (28.10) para X usando la descomposición LUP para A para resolver cada ecuación

$$AX_i = D e_i$$

por separado para  $X_i$ . Una vez que tenemos la descomposición LUP, podemos calcular cada una de las n columnas  $X_i$  en el tiempo  $,n^2/$ , y así podemos calcular X a partir de la posición de descomposición LUP de A en el tiempo  $,n^3/$ . Como podemos determinar la descomposición LUP de A en el tiempo  $,n^3/$ , podemos calcular la inversa  $A^{-1}$  de una matriz A en el tiempo  $,n^3/$ .

### Multiplicación de matrices e inversión de matrices

Ahora mostramos que las aceleraciones teóricas obtenidas para la multiplicación de matrices se traducen en aceleraciones para la inversión de matrices. De hecho, demostramos algo más fuerte: la inversión de matrices es equivalente a la multiplicación de matrices, en el siguiente sentido. Si  $M_n$  denota el tiempo para multiplicar dos matrices  $n \times n$ , entonces podemos invertir una matriz  $n \times n$  no singular en el tiempo  $O(M_n)$ . Además, si  $I_n$  denota el tiempo para invertir una matriz  $n \times n$  no singular, entonces podemos multiplicar dos matrices  $n \times n$  en el tiempo  $O(I_n)$ . Probamos estos resultados como dos teoremas separados.

#### Teorema 28.1 (La multiplicación no es más difícil que la inversión)

Si podemos invertir una matriz de  $n \times n$  en el tiempo  $I_n$ , donde  $I_n \leq n^2$  e  $I_n$  satisface la condición de regularidad  $I_n \leq O(I_n)$ , entonces podemos multiplicar dos matrices de  $n \times n$  en el tiempo  $O(I_n)$ .

Demostración Sean A y B matrices de  $n \times n$  cuyo producto matricial C deseamos calcular.

Definimos la matriz D  $3n \times 3n$  por

$$\begin{array}{ll} & \text{en un } 0 \\ \text{DD} & 0 \text{ en B} \\ & 0 \text{ En} \end{array}$$

El inverso de D es

$$\begin{array}{ll} & \text{en un } AB \\ \text{D1D} & _- 0 \text{ en B} \\ & 00 \text{En} \end{array}$$

y así podemos calcular el producto AB tomando la submatriz nn superior derecha de D1.

Podemos construir la matriz D en el tiempo  $\sqrt{n}$ , que es  $O\ln n$  porque suponemos que  $\ln D \leq \sqrt{n}$ , y podemos invertir D en el tiempo  $O(\ln n)^2 / D \ln n$ , por la condición de regularidad en  $\ln D$ . Así tenemos  $Mn/D = O(\ln n)^2 / D \ln n$ . ■

Nótese que  $\ln D$  satisface la condición de regularidad siempre que  $\ln D \geq c \lg n$  para cualquier constante  $c > 0$  y  $d > 0$ .

La prueba de que la inversión de matrices no es más difícil que la multiplicación de matrices se basa en algunas propiedades de las matrices definidas positivas simétricas que probaremos en la Sección 28.3.

**Teorema 28.2** (La inversión no es más difícil que la multiplicación)

Supongamos que podemos multiplicar dos nn matrices reales en el tiempo  $Mn^2$ , donde  $Mn^2 \leq Mn^k$  y  $Mn^k$  satisface las dos condiciones de regularidad  $Mn^k \leq C k^2 D$  para cualquier  $k$  en el rango  $0 \leq k \leq n$  y  $Mn^k \leq cM^k n^k$  para alguna constante  $c < 1$ . Entonces podemos calcular la inversa de cualquier matriz nn real no singular en el tiempo  $OMn^2$ .

**Demostración** Aquí demostramos el teorema para matrices reales. El ejercicio 28.2-6 le pide que generalice la prueba para matrices cuyas entradas son números complejos.

Podemos suponer que  $n$  es una potencia exacta de 2, ya que tenemos

$$\begin{array}{ccc} \text{un } 0 & \begin{matrix} 1 \\ \vdots \\ 0 \end{matrix} & A1 \text{ } 0 \\ 0 & \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} & \end{array}$$

para cualquier  $k > 0$ . Por lo tanto, al elegir  $k$  tal que  $n \leq 2^k$  es una potencia de 2, ampliamos la matriz a un tamaño que es la próxima potencia de 2 y obtenemos la respuesta deseada  $A1$  de la respuesta al problema ampliado. La primera condición de regularidad en  $Mn^k$  asegura que esta ampliación no haga que el tiempo de ejecución aumente en más de un factor constante.

Por el momento, supongamos que la matriz  $A$  de  $nn$  es simétrica y definida positiva. Dividimos cada uno de  $A$  y su inversa  $A1$  en cuatro  $n=2$   $n=2$  submatrices:

$$\begin{array}{c} \text{ANUNCIO} & \text{B TC} & \text{y A1D } \\ \text{CD} & & \end{array} \quad \begin{array}{c} \text{RT} \\ \text{ultravioleta} \end{array} \quad (28.11)$$

Entonces, si dejamos

$$\text{SDD CB1CT} \quad (28.12)$$

sea el complemento de Schur de A con respecto a B (veremos más sobre esta forma de complemento de Schur en la Sección

$$\begin{array}{c} \text{A1D } \\ \text{ultravioleta} \end{array} \quad \begin{array}{c} \text{RT} \\ \text{D} \end{array} \quad 28.3), \text{ tenemos } \begin{array}{c} \text{B1 C B1CTS1CB1} \\ \text{B1CTS1 S1CB1} \end{array} \quad \begin{array}{c} \text{S1} \end{array} \quad (28.13)$$

ya que AA1 D In, como puedes comprobar realizando la multiplicación de matrices. Como A es simétrico y definido positivo, los lemas 28.4 y 28.5 de la sección 28.3 implican que B y S son simétricos y definido positivo. Por el Lema 28.3 de la Sección 28.3, por lo tanto, existen los inversos B1 y S1 , y por el Ejercicio D.2-6, B1 y S1 son simétricos, de modo que .B1/T D B1 y .S1/T D S1. podemos calcular las submatrices R, T todasAllí , U y V de A1 como sigue, donde las matrices mencionadas son n=2

- n=2: 1. Forme las submatrices B, C, CT y D de A.
2. Calcule recursivamente el inverso B1 de B.
3. Calcule el producto matricial WD CB1 y luego calcule su transpuesta W T, que es igual a B1CT (mediante el ejercicio D.1-2 y .B1/T D B1).
4. Calcule el producto de matriz XDW CT, que es igual a CB1CT, y luego calcule la matriz SDDXDD CB1CT.
5. Calcule recursivamente el inverso S1 de S y establezca V en S1.
6. Calcular el producto matricial YD S1W , que es igual a S1CB1, y luego calcule su transpuesta Y T, que es igual a B1CTS1 (mediante el ejercicio D.1-2, .B1/T D B1 y .S1/T D S1). Establecer T a Y<sup>T</sup> y U a Y .
7. Calcule el producto matricial ZDW TY , que es igual a B1CTS1CB1, y conjunto R a B1 C Z.

Por lo tanto, podemos invertir una matriz definida positiva simétrica nn invirtiendo dos matrices n=2 n=2 en los pasos 2 y 5; realizar cuatro multiplicaciones de matrices n=2n=2 en los pasos 3, 4, 6 y 7; más un costo adicional de O.n2/ por extraer submatrices de A, insertar submatrices en A1 y realizar un número constante de sumas, restas y transposiciones en matrices n=2 n=2. Obtenemos la recurrencia

$$\text{En/ } 2I.n=2/ C 4M.n=2/ C O.n2/ D$$

$$2I.n=2/ C ,Mn// D OMn// :$$

La segunda línea se cumple porque la segunda condición de regularidad en el enunciado del teorema implica que  $4M.n=2/ < 2M.n/$  y porque suponemos que  $Mn/ D .n2 /$ . La tercera línea sigue porque la segunda condición de regularidad nos permite aplicar el caso 3 del teorema maestro (Teorema 4.1).

Queda por probar que podemos obtener el mismo tiempo asintótico de ejecución para la multiplicación de matrices que para la inversión de matrices cuando A es invertible pero no simétrica y definida positiva. La idea básica es que para cualquier matriz A no singular, la matriz ATA es simétrica (por el ejercicio D.1-2) y definida positiva (por el teorema D.6).

El truco, entonces, es reducir el problema de invertir A al problema de invertir ATA.

La reducción se basa en la observación de que cuando A es una matriz no singular nn, tenemos

$$A1 D .ATA/1AT ;$$

ya que ..ATA/1AT/A D .ATA/1.ATA/ D In y una matriz inversa es única.

Por lo tanto, podemos calcular A1 multiplicando primero AT por A para obtener ATA, luego invirtiendo la matriz definida positiva simétrica ATA usando el algoritmo anterior de divide y vencerás, y finalmente multiplicando el resultado por AT. Cada uno de estos tres pasos toma  $OMn//$  tiempo, y por lo tanto podemos invertir cualquier matriz no singular con entradas reales en  $OMn//$  tiempo. ■

La demostración del teorema 28.2 sugiere una forma de resolver la ecuación Ax D b usando la descomposición LU sin pivotar, siempre que A no sea singular. Multiplicamos ambos lados de la ecuación por AT, y obtenemos .ATA/x D ATb. Esta transformación no afecta la solución x, ya que AT es invertible, por lo que podemos factorizar la matriz definida positiva simétrica ATA calculando una descomposición LU. Luego usamos la sustitución hacia adelante y hacia atrás para resolver x con el lado derecho ATb. Aunque este método es teóricamente correcto, en la práctica el procedimiento LUP-DECOMPOSITION funciona mucho mejor. La descomposición LUP requiere menos operaciones aritméticas por un factor constante y tiene propiedades numéricas algo mejores.

## Ejercicios

### 28.2-1

Sea  $Mn/$  el tiempo para multiplicar dos matrices de nn, y sea  $Sn/$  el tiempo requerido para elevar al cuadrado una matriz de nn. Muestre que multiplicar y elevar al cuadrado las matrices tienen esencialmente la misma dificultad: un algoritmo de multiplicación de matrices de  $Mn/-tiempo$  implica un algoritmo de elevación al cuadrado de  $OMn/-tiempo$ , y un algoritmo de elevación de  $Sn/-tiempo$  al cuadrado implica una matriz de  $OSn/-tiempo$ . algoritmo de multiplicación.

**28.2-2**

Sea  $M_n$  el tiempo para multiplicar dos matrices de  $n \times n$ , y sea  $L_n$  el tiempo para calcular la descomposición LUP de una matriz de  $n \times n$ . Muestre que multiplicar matrices y calcular las descomposiciones LUP de matrices tienen esencialmente la misma dificultad: un algoritmo de multiplicación de matrices  $M_n$ -tiempo implica un algoritmo de descomposición LUP  $O(M_n)$ -tiempo, y un algoritmo de descomposición LUP  $L_n$ -tiempo implica un algoritmo de multiplicación de matrices de tiempo  $O(L_n)$ .

**28.2-3**

Sea  $M_n$  el tiempo para multiplicar dos matrices de  $n \times n$  y sea  $D_n$  el tiempo necesario para encontrar el determinante de una matriz de  $n \times n$ . Muestre que la multiplicación de matrices y el cálculo del determinante tienen esencialmente la misma dificultad: un algoritmo de multiplicación de matrices de  $M_n$ -tiempo implica un algoritmo de determinante de tiempo  $O(M_n)$ , y un algoritmo de determinante de tiempo  $D_n$ -implica un tiempo de  $O(D_n)$  algoritmo de multiplicación de matrices.

**28.2-4**

Sea  $M_n$  el tiempo para multiplicar dos matrices booleanas de  $n \times n$ , y sea  $T_{n,n}$  el tiempo para encontrar la clausura transitiva de una matriz booleana de  $n \times n$ . (Consulte la Sección 25.2.) Muestre que un algoritmo de multiplicación de matriz booleana  $M_n$ -tiempo implica un algoritmo de cierre transitivo  $O(M_n \lg n)$ -tiempo, y un algoritmo de cierre transitivo  $T_{n,n}$ -tiempo implica una matriz booleana  $O(T_{n,n})$ -tiempo. algoritmo de multiplicación.

**28.2-5**

¿Funciona el algoritmo de inversión de matriz basado en el teorema 28.2 cuando los elementos de la matriz se extraen del campo de los enteros módulo 2? Explicar.

**28.2-6 ?**

Generalice el algoritmo de inversión de matrices del Teorema 28.2 para manejar matrices de números complejos y demuestre que su generalización funciona correctamente. (Sugerencia: en lugar de la transpuesta de  $A$ , use la transpuesta conjugada  $A^*$ , que obtiene de la transpuesta de  $A$  reemplazando cada entrada con su complejo conjugado. En lugar de matrices simétricas, considere matrices hermitianas , que son matrices  $A$  tales que  $AD A^*$ .)

---

## 28.3 Matrices definidas positivas simétricas y aproximación por mínimos cuadrados

Las matrices definidas positivas simétricas tienen muchas propiedades interesantes y deseables. Por ejemplo, no son singulares y podemos realizar la descomposición LU de ellos sin tener que preocuparnos por dividir por 0. En esta sección, vamos a

probar varias otras propiedades importantes de las matrices definidas positivas simétricas y mostrar una aplicación interesante al ajuste de curvas mediante una aproximación de mínimos cuadrados.

La primera propiedad que probamos es quizás la más básica.

#### Lema 28.3

Cualquier matriz definida positiva es no singular.

Prueba Suponga que una matriz A es singular. Entonces, por el Corolario D.3, existe un vector x distinto de cero tal que  $Ax = 0$ . Por lo tanto,  $x^T Ax = 0$ , y A no puede ser positivo definido. ■

La prueba de que podemos realizar la descomposición LU en una matriz definida positiva simétrica A sin dividir por 0 es más compleja. Comenzamos demostrando propiedades sobre ciertas submatrices de A. Definimos la k-ésima submatriz principal de A como la matriz  $A_k$  que consiste en la intersección de las primeras k filas y las primeras k columnas de A.

#### Lema 28.4 Si A

es una matriz definida positiva simétrica, entonces toda submatriz principal de A es simétrica y definida positiva.

La prueba de que cada submatriz principal  $A_k$  es simétrica es obvia. Para probar que  $A_k$  es positivo-definido, asumimos que no lo es y derivamos una contradicción. Si  $A_k$  no es definido positivo, entonces existe un k-vector  $x_k \neq 0$  tal que  $x_k^T A_k x_k \leq 0$ . Sea A nn, y

$$A_k B^T$$

ANUNCIO

antes de Cristo

(28.14)

para las submatrices B (que es  $.nk/k$ ) y C (que es  $.nk/.nk/$ ). Defina el vector n x D .  $x^T$

$k = 0 / T$ , donde nk 0s siguen a  $x_k$ . Entonces nosotros tenemos

$$x^T A_k B^T = x^T A_k x_k = 0$$

$$\begin{matrix} & \\ & akxk \\ & \text{caja} \end{matrix}$$

$$D . x^T A_k x_k = 0$$

;

lo que contradice que A sea positivo-definido. ■

Pasamos ahora a algunas propiedades esenciales del complemento de Schur. Sea  $A$  una matriz definida positiva simétrica, y sea  $A_k$  una submatriz principal  $k \times k$  de  $A$ . Divida  $A$  una vez más de acuerdo con la ecuación (28.14). Generalizamos la ecuación (28.9) para definir el complemento de Schur  $S$  de  $A$  con respecto a  $A_k$  como SDC

BA1 BT k : (28.15)

(Por el Lema 28.4,  $A_k$  es simétrico y positivo-definido; por lo tanto,  $A_1$  existe por el Lema 28.3, y  $S$  está bien definido.) Nótese que nuestra definición anterior (28.9) del complemento de Schur es consistente con la ecuación (28.15), al hacer  $k = 1$ .

El siguiente lema muestra que las matrices de complemento de Schur de matrices definidas positivas simétricas son ellas mismas simétricas y definidas positivas. Usamos este resultado en el Teorema 28.2, y necesitamos su corolario para demostrar que la descomposición LU es correcta para matrices definidas positivas simétricas.

Lema 28.5 (lema del complemento de Schur)

Si  $A$  es una matriz definida positiva simétrica y  $A_k$  es una submatriz principal  $k \times k$  de  $A$ , entonces el complemento de Schur  $S$  de  $A$  con respecto a  $A_k$  es simétrico y definido positivamente.

Prueba Como A es simétrica, también lo es la submatriz C. Por el ejercicio D.2-6, el producto  $BA_1 BT$  es simétrico, y por el ejercicio D.1-1, S es simétrica.

Queda por demostrar que  $S$  es definido positivo. Considere la partición de  $A$  dada en la ecuación (28.14). Para cualquier vector  $x$  distinto de cero, tenemos  $x^T A x > 0$  por la suposición de que  $A$  es definida positiva. Descompongamos  $x$  en dos subvectores  $y$  y  $z$  compatibles con  $A_k$  y  $C$ , respectivamente. Como  $A_1$  existe,

por magia matricial. (Compruebe multiplicando). Esta última ecuación equivale a "completar el cuadrado" de la forma cuadrática. (Consulte el ejercicio 28-3-2.)

Dado que  $x^T A x > 0$  se cumple para cualquier  $x$  distinto de cero, escogamos cualquier  $y$  distinto de cero y luego elijamos  $y \in D A_1 B T'$ , lo que hace que desaparezca el primer término de la

ecuación (28.16), dejando 'TC

BA1 BT/’ D ‘TS’ como el valor de la expresión. Para cualquier  $\epsilon > 0$ , tenemos por lo tanto ‘TS’ D  $xT\Delta x > 0$ , y por lo tanto S es definida positiva.

## Corolario 28.6 La

descomposición LU de una matriz definida positiva simétrica nunca provoca una división por 0.

Demostración Sea A una matriz definida positiva simétrica. Probaremos algo más fuerte que el enunciado del corolario: todo pivote es estrictamente positivo. El primer pivote es  $a_{11}$ . Sea  $e_1$  el primer vector unitario, del cual obtenemos  $a_{11} D e_1^T 1 A e_1 > 0$ .

Dado que el primer paso de la descomposición LU produce el complemento de Schur de A con respecto a  $A \setminus a_{11}$ , el Lema 28.5 implica por inducción que todos los pivotes son positivos. ■

## Aproximación por mínimos cuadrados

Una aplicación importante de las matrices definidas positivas simétricas surge al ajustar curvas a conjuntos dados de puntos de datos. Supongamos que nos dan un conjunto de m puntos de datos

$x_1; y_1/; x_2; y_2/; \dots; x_m; y_m/;$

donde sabemos que las  $y_i$  están sujetas a errores de medición. Nos gustaría determinar una función  $F(x)$  tal que los errores de aproximación

$$\sum_i (F(x_i) - y_i)^2 \quad (28.17)$$

son pequeños para  $i = 1; 2; \dots; m$ . La forma de la función  $F$  depende del problema en cuestión. Aquí, asumimos que tiene la forma de una suma ponderada linealmente,

$$F(x) = \sum_{j=1}^n c_j f_j(x);$$

donde el número de sumandos  $n$  y las funciones de base específicas  $f_j$  se eligen en función del conocimiento del problema en cuestión. Una opción común es  $f_j(x) = x^{j-1}$ , lo que significa que

$$F(x) = c_1 + c_2 x + c_3 x^2 + \dots + c_n x^{n-1}$$

es un polinomio de grado  $n$  en  $x$ . Así, dados  $m$  puntos de datos  $x_1; y_1/; x_2; y_2/; \dots; x_m; y_m/$ , deseamos calcular  $n$  coeficientes  $c_1; c_2; \dots; c_n$  que minimizan los errores de aproximación  $1; 2; \dots; m$ .

Al elegir  $n \leq m$ , podemos calcular cada  $y_i$  exactamente en la ecuación (28.17). Sin embargo, un grado  $F$  tan alto "se ajusta al ruido" así como a los datos y generalmente da resultados deficientes cuando se usa para predecir y para valores de  $x$  nunca antes vistos. Por lo general, es mejor elegir  $n$  significativamente menor que  $m$  y esperar que al elegir bien los coeficientes  $c_j$ , podamos obtener una función  $F$  que encuentre los patrones significativos en los puntos de datos sin prestar demasiada atención al ruido. algo de teoría

existen principios para elegir  $n$ , pero están más allá del alcance de este texto. En cualquier caso, una vez que elegimos un valor de  $n$  menor que  $m$ , terminamos con un conjunto de ecuaciones sobredeterminado cuya solución deseamos aproximar. Ahora mostramos cómo hacerlo.

Dejar

$$\begin{array}{c} f_1.x_2/ f_2.x_2/ \cdots f_n.x_2/ \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ f_1.x_1/ f_2.x_1/ \cdots f_n.x_1/ \end{array}$$

denote la matriz de valores de las funciones base en los puntos dados; es decir,  $a_{ij} = f_j(x_i)$ . Sea  $c = [c_1 \ c_2 \ \dots \ c_m]^T$  el  $n$ -vector de coeficientes deseado. Entonces,

$$\begin{array}{c} f_1.x_2/ f_2.x_2/ \cdots f_n.x_2/ \qquad \qquad \qquad c_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \qquad \qquad \qquad \vdots \\ f_1.x_1/ f_2.x_1/ \cdots f_n.x_1/ \qquad \qquad \qquad c_m \\ \text{Ac} = [f_1.x_1 \ f_2.x_1 \ \cdots \ f_n.x_1] \quad \cdots \quad c_m \\ F.x_2/ \\ \vdots \\ F.x_m/ \end{array}$$

es el  $m$ -vector de "valores pronosticados" para  $y$ . De este modo,

$$D = Ac - y$$

es el  $m$ -vector de errores de aproximación.

Para minimizar los errores de aproximación, elegimos minimizar la norma del error que vector  $D$ , nos da una solución de mínimos cuadrados, ya que

$$\|D\|_2^2 = \sum_{i=1}^m \|D_i\|^2$$

Porque

$$\|D\|_2^2 = \sum_{i=1}^m \|D_i\|^2 = \sum_{i=1}^m (y_i - \sum_{j=1}^n a_{ij} c_j)^2$$

podemos minimizar  $\|D\|_2^2$  diferenciando  $\|D\|_2^2$  con respecto a cada  $c_j$  y luego estableciendo el resultado en 0:

$$\frac{d \ k k_2}{d c_k} D \sum_{i=1}^n a_{ii} c_i y_i = a_{ik} D \quad (28.18)$$

Las  $n$  ecuaciones (28.18) para  $k = 1; 2; \dots; n$  son equivalentes a la ecuación matricial única

$$A^T A c = A^T b$$

o, de manera equivalente (usando el ejercicio D.1-2), para

$$A^T A c = b$$

lo que implica

$$ATAc = b \quad (28.19)$$

En estadística, esto se llama la ecuación normal. La matriz  $ATA$  es simétrica por el ejercicio D.1-2, y si  $A$  tiene rango de columna completo, entonces por el teorema D.6,  $ATA$  también es definida positiva. Por lo tanto,  $.ATA^{-1}$  existe y la solución a la ecuación (28.19) es

$$c = ATA^{-1}AT^T b = AC^{-1}A^T b \quad (28.20)$$

donde la matriz  $AC = ATA^{-1}AT^T$  es la pseudoinversa de la matriz  $A$ . La pseudoinversa naturalmente generaliza la noción de una matriz inversa al caso en el que  $A$  no es cuadrada. (Compare la ecuación (28.20) como la solución aproximada de  $Ac = b$  y con la solución  $A^{-1}b$  como la solución exacta de  $Ax = b$ ).

Como ejemplo de producción de un ajuste de mínimos cuadrados, supongamos que tenemos cinco puntos de datos

$$\begin{aligned} x_1 &= 1; y_1 = 2; \\ x_2 &= 2; y_2 = 1; \\ x_3 &= 3; y_3 = 2; \\ x_4 &= 4; y_4 = 3; \\ x_5 &= 5; y_5 = 5; \end{aligned}$$

se muestra como puntos negros en la Figura 28.3. Deseamos ajustar estos puntos con un polinomio cuadrático

$$F(x) = c_1 + c_2 x + c_3 x^2$$

Empezamos con la matriz de valores de funciones base

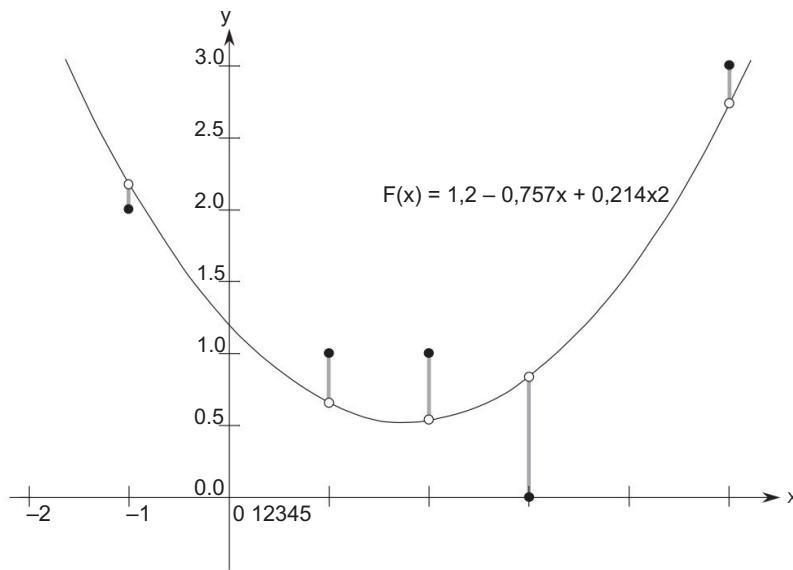


Figura 28.3 El ajuste por mínimos cuadrados de un polinomio cuadrático al conjunto de cinco puntos de datos f.1; 2/.1; 1/.2; 1/.3; 0/.5; 3/g. Los puntos negros son los puntos de datos y los puntos blancos son sus valores estimados predichos por el polinomio  $F(x) = 1.2 - 0.757x + 0.214x^2$ , el polinomio cuadrático que minimiza la suma de los errores cuadráticos. Cada línea sombreada muestra el error de un punto de datos.

$1 \times 1 \times 2 \_ \_ 1$	1	1	1
$1 \times 2 \times 2 \_ \_ 2$	1	11	
ANUNCIO	1	24	
$1 \times 3 \times 2 \_ \_ 3$	1	24	
$1 \times 4 \times 2 \_ \_ 4$	1	39	
$1 \times 5 \times 2 \_ \_ 5$	1	525	

cuyo pseudoinverso es

$$\begin{array}{ccccccccc} & 0:500 & 0:300 & 0:200 & 0:100 & 0:100 & 0:388 & 0:093 & 0:190 \\ \text{CA D} & 0:193 & 0:088 & 0:060 & 0:036 & 0:048 & 0:036 & 0:060 & \end{array}$$

Multiplicando y por AC, obtenemos el vector de coeficientes

$$\begin{array}{l} 1:200 \\ \text{cd} \quad 0:757 \\ \quad \quad \quad ; \\ \quad \quad \quad 0:214 \end{array}$$

que corresponde al polinomio cuadrático

F .x/ D 1:200 0:757x C 0:214x2

como la cuadrática que más se ajusta a los datos dados, en un sentido de mínimos cuadrados.

Como cuestión práctica, resolvemos la ecuación normal (28.19) multiplicando y por  $A^T$  y luego encontrando una descomposición LU de  $ATA$ . Si  $A$  tiene rango completo, se garantiza que la matriz  $ATA$  no es singular, porque es simétrica y definida positiva. (Consulte el ejercicio D.1-2 y el teorema D.6).

### Ejercicios

#### 28.3-1

Demuestre que todo elemento diagonal de una matriz definida positiva simétrica es positivo.

#### 28.3-2

Deja que  $AD = \begin{matrix} abc \\ \text{sea una matriz definida positiva simétrica } 2 \end{matrix}$ . Demostrar que es

el determinante  $ac - b^2$  es positivo al “completar el cuadrado” de manera similar a la utilizada en la demostración del Lema 28.5.

#### 28.3-3

Demuestre que el elemento máximo en una matriz definida positiva simétrica se encuentra en la diagonal.

#### 28.3-4

Demostrar que el determinante de cada submatriz principal de una matriz definida positiva simétrica es positivo.

#### 28.3-5

Sea  $A_k$  la  $k$ -ésima submatriz principal de una matriz definida positiva simétrica  $A$ .

Demuestre que  $\det.A_k/ = \det.A_{k-1}/$  es el pivote  $k$ -ésimo durante la descomposición LU, donde, por convención,  $\det.A_0/ = 1$ .

#### 28.3-6

Encuentre la función de la forma

F .x/ D c1 C c2x lg x C c3ex

que es el mejor ajuste de mínimos cuadrados a los puntos de datos

.1; 1/; .2; 1/; .3; 3/; .4; 8/ :

## 28.3-7

Muestre que la pseudoinversa  $A^+C$  satisface las siguientes cuatro ecuaciones:

$$\begin{aligned} & D \cdot A \cdot A \cdot A \cdot A^+ \\ & A \cdot C \cdot A \cdot C \cdot A \cdot C \cdot A^+ \\ & A \cdot A \cdot C^T \cdot D \\ & A \cdot A \cdot C \cdot A \cdot C \cdot A^T \cdot D \cdot A \cdot C \end{aligned}$$

## Problemas

## 28-1 Sistemas tridiagonales de ecuaciones lineales

Considere la matriz tridiagonal

$$\begin{matrix} 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 0 & 0 \end{matrix} \text{ de } C$$

- a. Encuentre una descomposición LU de  $A$ .
- b. Resolver la ecuación  $Ax = 11111$  mediante el uso de adelante y atrás sub constitución.
- C. Encuentre el inverso de  $A$ .
- d. Muestre cómo, para cualquier matriz  $A$  tridiagonal definida positiva simétrica  $n \times n$  y cualquier  $n$ -vector  $b$ , resolver la ecuación  $Ax = b$  en tiempo  $O(n^3)$  realizando una descomposición LU. Argumente que cualquier método basado en la formación de  $A^T$  es asintóticamente más costoso en el peor de los casos.
- mi. Muestre cómo, para cualquier  $n \times n$  no singular, matriz tridiagonal  $A$  y cualquier  $n$ -vector  $b$ , resolver la ecuación  $Ax = b$  en tiempo  $O(n^3)$  realizando una descomposición LUP.

## 28-2 Splines

Un método práctico para interpolar un conjunto de puntos con una curva es usar splines cúbicos. Nos dan un conjunto  $(x_i, y_i) \in W$  para  $i = 0, 1, \dots, n$  de  $n + 1$  pares de valor de punto, donde  $x_0 < x_1 < \dots < x_n$ . Deseamos ajustar una curva cúbica por partes (spline)  $f(x)$  a los puntos. Es decir, la curva  $f(x)$  está formada por  $n$  polinomios cúbicos  $f_i(x)$  para  $i = 0, 1, \dots, n - 1$ , donde si  $x$  cae en

el rango  $x_i \leq x \leq x_{i+1}$ , entonces el valor de la curva viene dado por  $f(x) = f_i(x)$ .

Los puntos  $x_i$  en los que se “pegan” los polinomios cúbicos se denominan nudos.

Por simplicidad, supondremos que  $x_i \neq D_0, D_1, \dots, D_n$ . Para

norte.

asegurar la continuidad de  $f(x)$ , requerimos que

$$f(x_i) = f_i(0), \quad f'(x_i) = f'_i(0)$$

$$y_i = f(x_i)$$

para  $i = 0, 1, \dots, n-1$ . Para asegurar que  $f(x)$  sea lo suficientemente suave, también insistimos en que la primera derivada sea continua en cada nudo:

$$f''(x_i) = f''_i(0)$$

$$y''_i = f''(x_i)$$

- a. Suponga que para  $i = 0, 1, \dots, n$ , se nos dan no sólo los pares punto-valor  $(x_i, y_i)$  sino también las primeras derivadas  $D_i$  y el coeficiente  $a_i$ ,  $b_i$ ,  $c_i$  y  $d_i$  en términos del nudo. Expresar cada uno de los valores  $y_i$ ,  $y'_i$ ,  $y''_i$ ,  $D_i$  y  $D'_i$ .

(Recuerde que  $x_i = D_i$ .) ¿Con qué rapidez podemos calcular los coeficientes  $a_i, b_i, c_i, d_i$  a partir de los pares de valores puntuales y las primeras derivadas?

Queda la cuestión de cómo elegir las primeras derivadas de  $f(x)$  en los nudos.

Un método es requerir que las segundas derivadas sean continuas en los nudos:

$$f''(x_i) = f''_i(0)$$

1;  $\dots$ ;  $f''(x_n) = f''_n(0)$ . En el primer y último nudo, suponemos que  $f''(x_0) = f''(x_n) = 0$ ; estas suposiciones hacen que  $f(x)$  sea natural cónica.

- b. Use las restricciones de continuidad en la segunda derivada para mostrar que para  $i = 0, 1, 2, \dots, n-1$ ,

$$D_i^2 + 4D_i C_i + D_i C_i^2 = 3y''_i : \quad (28.21)$$

C. Muestra esa

$$2D_0 C_0 D_1 C_1 3.y''_0 ; \quad D_0 C_0 2D_1 C_1 \quad (28.22)$$

$$D_3.y''_n = 0 : \quad (28.23)$$

- d. Reescriba las ecuaciones (28.21)–(28.23) como una ecuación matricial que involucre al vector  $\mathbf{y} = [y_0, y_1, \dots, y_n]^T$  de incógnitas. ¿Qué atributos tiene la matriz en su ecuación?

mi. Argumente que un spline cúbico natural puede interpolar un conjunto de  $n+1$  pares de valores puntuales en el tiempo  $[0, 1]$  (vea el problema 28-1).

F. Mostrar cómo determinar un spline cúbico natural que interpola un conjunto de  $n \in \mathbb{C}^1$  puntos  $(x_i, y_i)$  satisfaciendo  $x_0 < x_1 < \dots < x_n$ , incluso cuando  $x_i$  no es necesariamente igual a  $i$ . ¿Qué ecuación matricial debe resolver su método y qué tan rápido se ejecuta su algoritmo?

---

### Notas del capítulo

Muchos textos excelentes describen el cálculo numérico y científico con mucho más detalle del que tenemos aquí. Los siguientes son especialmente legibles: George y Liu [132], Golub y Van Loan [144], Press, Teukolsky, Vetterling y Flannery [283, 284] y Strang [323, 324].

Golub y Van Loan [144] analizan la estabilidad numérica. Muestran por qué  $\det(A)$  no es necesariamente un buen indicador de la estabilidad de una matriz  $A$ , proponiendo en su lugar utilizar  $\|A\|_1 \|A^{-1}\|_1$ , donde  $\|A\|_1 = \max_{1 \leq i \leq n} \sum_{j=1}^m |a_{ij}|$ . También abordan la cuestión de cómo calcular este valor sin calcular realmente  $A^{-1}$ .

La eliminación gaussiana, en la que se basan las descomposiciones LU y LUP, fue el primer método sistemático para resolver sistemas de ecuaciones lineales. También fue uno de los primeros algoritmos numéricos. Aunque se conocía antes, su descubrimiento se atribuye comúnmente a Carl Friedrich Gauss (1777-1855). En su famoso artículo [325], Strassen mostró que una matriz  $n \times n$  puede invertirse en  $O(n \lg 7)$  tiempo. Winograd [358] demostró originalmente que la multiplicación de matrices no es más difícil que la inversión de matrices, y lo contrario se debe a Aho, Hopcroft y Ullman [5].

Otra descomposición matricial importante es la descomposición en valores singulares, o SVD. El SVD factoriza una matriz  $A \in \mathbb{R}^{m \times n}$  en la matriz  $A = Q_1 \Sigma Q_2^\top$ , donde  $\Sigma$  es una matriz diagonal con valores distintos de cero solo en la diagonal,  $Q_1$  es  $m \times m$  con columnas mutuamente ortonormales y  $Q_2$  es  $n \times n$ , también con columnas mutuamente ortonormales.

Dos vectores son ortonormales si su producto interno es 0 y cada vector tiene una norma de 1. Los libros de Strang [323, 324] y Golub y Van Loan [144] contienen buenos tratamientos de la SVD.

Strang [324] tiene una excelente presentación de matriz definida positiva simétrica y del álgebra lineal en general.

---

## 29

## Programación lineal

Muchos problemas toman la forma de maximizar o minimizar un objetivo, dados los recursos limitados y las limitaciones que compiten entre sí. Si podemos especificar el objetivo como una función lineal de ciertas variables, y si podemos especificar las restricciones sobre los recursos como igualdades o desigualdades en esas variables, entonces tenemos un problema de programación lineal. Los programas lineales surgen en una variedad de aplicaciones prácticas. Comenzamos estudiando una aplicación en la política electoral.

### Un problema político

Suponga que usted es un político que intenta ganar unas elecciones. Su distrito tiene tres tipos diferentes de áreas: urbana, suburbana y rural. Estas áreas tienen, respectivamente, 100.000, 200.000 y 50.000 votantes registrados. Aunque no todos los votantes registrados acuden realmente a las urnas, usted decide que, para gobernar de manera efectiva, le gustaría que al menos la mitad de los votantes registrados en cada una de las tres regiones votaran por usted. Eres honorable y nunca considerarías apoyar políticas en las que no crees. Sin embargo, te das cuenta de que ciertos temas pueden ser más efectivos para ganar votos en ciertos lugares. Sus principales problemas son la construcción de más carreteras, el control de armas, los subsidios agrícolas y un impuesto a la gasolina dedicado a mejorar el transporte público.

De acuerdo con la investigación del personal de su campaña, puede estimar cuántos votos gana o pierde de cada segmento de la población gastando \$1,000 en publicidad en cada tema. Esta información aparece en la tabla de la Figura 29.1. En esta tabla, cada entrada indica la cantidad de miles de votantes urbanos, suburbanos o rurales que se ganarían si gastaran \$ 1,000 en publicidad en apoyo de un tema en particular. Las entradas negativas denotan votos que se perderían. Tu tarea es calcular la cantidad mínima de dinero que necesitas gastar para ganar 50 000 votos urbanos, 100 000 votos suburbanos y 25 000 votos rurales.

Podría, por ensayo y error, idear una estrategia que gane el número requerido de votos, pero la estrategia que se le ocurra podría no ser la menos costosa.

Por ejemplo, podrías dedicar \$20 000 de publicidad a la construcción de caminos, \$0 al control de armas, \$4 000 a subsidios agrícolas y \$9 000 a un impuesto a la gasolina. En este caso, usted

política	urbano	Suburbano	Rural
construir caminos	2	5	3
control de armas		2	5
subsidios agrícolas	8	0	10
impuesto a la gasolina	0 10	0	2

Figura 29.1 Los efectos de las políticas en los votantes. Cada entrada describe la cantidad de miles de votantes urbanos, suburbanos o rurales que podrían ganarse gastando \$ 1,000 en apoyo publicitario de una política sobre un tema en particular. Las entradas negativas denotan votos que se perderían.

ganaría 20.2/C0.8/C4.0/C9.10/ D 50 mil votos urbanos, 20.5/C0.2/C 4.0/C9.0/ D 100 mil votos suburbanos y 20.3/C0.5/C4. 10/C9.2/ D 82 mil votos rurales. Ganaría el número exacto de votos deseados en las áreas urbanas y suburbanas y votos más que suficientes en el área rural. (De hecho, en el área rural, recibiría más votos que votantes). Para obtener estos votos, habría pagado 20 C 0 C 4 C 9 D 33 mil dólares de publicidad.

Naturalmente, puede preguntarse si esta estrategia es la mejor posible. Es decir, ¿podrías lograr tus objetivos gastando menos en publicidad? Prueba y error adicionales pueden ayudarlo a responder esta pregunta, pero ¿no preferiría tener un método sistemático para responder tales preguntas? Para desarrollar uno, formularemos esta pregunta matemáticamente. Introducimos 4 variables:

x1 es el número de miles de dólares gastados en publicidad sobre la construcción de carreteras, x2 es el número de miles de dólares gastados en publicidad sobre el control de armas, x3 es el número de miles de dólares gastados en publicidad sobre subsidios agrícolas, y

x4 es el número de miles de dólares gastados en publicidad sobre un impuesto a la gasolina.

Podemos escribir el requisito de ganar al menos 50.000 votos urbanos como

$$2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \quad (29.1)$$

De manera similar, podemos escribir los requisitos de que ganamos al menos 100 000 votos suburbanos y 25 000 votos rurales como

$$5x_1 + 2x_2 + 0x_3 + 10x_4 \geq 100 \quad (29.2)$$

y

$$3x_1 + 5x_2 + 10x_3 + 2x_4 \geq 25 \quad (29.3)$$

Cualquier ajuste de las variables x1; x2; x3; x4 que satisface las desigualdades (29.1)–(29.3) produce una estrategia que gana un número suficiente de cada tipo de voto. Con el fin de

mantener los costos lo más bajos posible, le gustaría minimizar la cantidad gastada en publicidad. Es decir, desea minimizar la expresión

$$x_1 \leq x_2 \leq x_3 \leq x_4 : \quad (29.4)$$

Aunque la publicidad negativa a menudo ocurre en las campañas políticas, no existe la publicidad de costo negativo. En consecuencia, requerimos que

$$x_1 \geq 0; x_2 \geq 0; x_3 \geq 0; y \leq x_4 \quad (29.5)$$

Combinando las desigualdades (29.1)–(29.3) y (29.5) con el objetivo de minimizar (29.4), obtenemos lo que se conoce como “programa lineal”. Formateamos este problema

como

$$\text{minimizar} \quad x_1 \leq x_2 \leq x_3 \leq x_4 \quad (29.6)$$

sujeto a

$$2x_1 \leq 8x_2 \leq 0x_3 \leq 10x_4 \leq 5x_1 \leq 2x_2 \leq 0x_3 \leq 50 \quad (29.7)$$

$$C \leq 0x_4 \leq 3x_1 \leq 2x_4 \leq 100 \quad (29.8)$$

$$5x_2 \leq C \leq 10x_3 \leq 25 \quad (29.9)$$

$$x_1; x_2; x_3; x_4 \leq 0 \quad (29.10)$$

La solución de este programa lineal produce su estrategia óptima.

### Programas lineales generales

En el problema general de programación lineal, deseamos optimizar una función lineal sujeta a un conjunto de desigualdades lineales. Dado un conjunto de números reales  $a_1; a_2; \dots; a_n$  y un conjunto de variables  $x_1; x_2; \dots; x_n$ , definimos una función lineal  $f$  en esas variables por

$$f(x_1; x_2; \dots; x_n) = a_1x_1 + a_2x_2 + \dots + a_nx_n \quad \text{a } j : \quad (29.11)$$

Si  $b$  es un número real y  $f$  es una función lineal, entonces la ecuación

$$f(x_1; x_2; \dots; x_n) = b \quad \text{segundo}$$

es una igualdad lineal y las desigualdades

$$f(x_1; x_2; \dots; x_n) \leq b$$

y

$$f(x_1; x_2; \dots; x_n) \geq b$$

son desigualdades lineales. Usamos el término general restricciones lineales para denotar igualdades lineales o desigualdades lineales. En programación lineal, no permitimos desigualdades estrictas. Formalmente, un problema de programación lineal es el problema de minimizar o maximizar una función lineal sujeta a un conjunto finito de restricciones lineales. Si vamos a minimizar, entonces llamamos al programa lineal un programa lineal de minimización, y si vamos a maximizar, entonces llamamos al programa lineal un programa lineal de maximización.

El resto de este capítulo cubre cómo formular y resolver programas lineales. Aunque se han desarrollado varios algoritmos de tiempo polinomial para programación lineal, no los estudiaremos en este capítulo. En su lugar, estudiaremos el algoritmo simplex, que es el algoritmo de programación lineal más antiguo. El algoritmo simplex no se ejecuta en tiempo polinomial en el peor de los casos, pero es bastante eficiente y se usa ampliamente en la práctica.

#### Una visión general de la programación lineal

Para describir propiedades y algoritmos para programas lineales, encontramos conveniente expresarlos en formas canónicas. Usaremos dos formas, estándar y slack, en este capítulo. Los definiremos con precisión en el apartado 29.1. Informalmente, un programa lineal en forma estándar es la maximización de una función lineal sujeta a desigualdades lineales, mientras que un programa lineal en forma holgada es la maximización de una función lineal sujeta a igualdades lineales. Usualmente usaremos la forma estándar para expresar programas lineales, pero encontramos que es más conveniente usar la forma holgada cuando describimos los detalles del algoritmo simplex. Por ahora, restringimos nuestra atención a maximizar una función lineal en  $n$  variables sujetas a un conjunto de  $m$  desigualdades lineales.

Consideremos primero el siguiente programa lineal con dos variables:

$$\text{maximizar } x_1 + x_2 \text{ sujeto a} \quad (29.11)$$

$$4x_1 + x_2 \leq 8 \quad (29.12)$$

$$2x_1 + 5x_2 \leq 10 \quad (29.13)$$

$$- \quad \quad \quad 2 \quad (29.14)$$

$$x_1, x_2 \geq 0 \quad (29.15)$$

Cualquier ajuste de las variables  $x_1$  y  $x_2$  que satisfaga todas las restricciones (29.12)–(29.15) lo llamamos solución factible del programa lineal. Si graficamos las restricciones en  $x_1$ - $x_2$ -sistema de coordenadas cartesianas, como en la figura 29.2(a), vemos

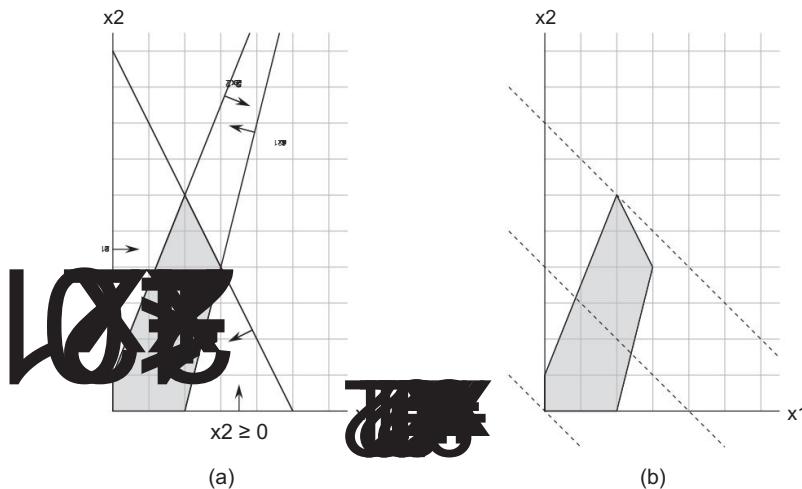


Figura 29.2 (a) El programa lineal dado en (29.12)–(29.15). Cada restricción está representada por una línea y una dirección. La intersección de las restricciones, que es la región factible, está sombreada. (b) Las líneas punteadas muestran, respectivamente, los puntos para los cuales el valor objetivo es 0, 4 y 8. La solución óptima del programa lineal es  $x_1 = 2$  y  $x_2 = 6$  con valor objetivo 8.

que el conjunto de soluciones factibles (sombreadas en la figura) forma una región convexa<sup>1</sup> en el espacio bidimensional. Llamamos a esta región convexa la región factible y la función que deseamos maximizar la función objetivo. Conceptualmente, podríamos evaluar la función objetivo  $x_1 + x_2$  en cada punto de la región factible; llamamos valor objetivo al valor de la función objetivo en un punto particular. Entonces podríamos identificar un punto que tenga el valor objetivo máximo como una solución óptima.

Para este ejemplo (y para la mayoría de los programas lineales), la región factible contiene un número infinito de puntos, por lo que necesitamos determinar una forma eficiente de encontrar un punto que logre el valor objetivo máximo sin evaluar explícitamente la función objetivo en cada punto en la región factible.

En dos dimensiones, podemos optimizar a través de un procedimiento gráfico. El conjunto de puntos para los cuales  $x_1 + x_2 \leq z$ , para cualquier  $z$ , es una recta con pendiente 1. Si graficamos  $x_1 + x_2 = 0$ , obtenemos la recta con pendiente 1 a través del origen, como en la figura 29.2(b). La intersección de esta recta y la región factible es el conjunto de soluciones factibles que tienen un valor objetivo de 0. En este caso, esa intersección de la recta con la región factible es el único punto  $(0, 0)$ . Más generalmente, para cualquier  $z$ , la intersección

---

<sup>1</sup>Una definición intuitiva de una región convexa es que cumple el requisito de que para dos puntos cualquiera de la región, todos los puntos en un segmento de línea entre ellos también están en la región.

de la recta  $x_1 \leq x_2 \leq D$  y la región factible es el conjunto de soluciones factibles que tienen valor objetivo  $C$ . La figura 29.2(b) muestra las rectas  $x_1 \leq x_2 \leq 0$ ,  $x_1 \leq x_2 \leq 4$  y  $x_1 \leq x_2 \leq 8$ . Como la región factible de la figura 29.2 está acotada, debe haber algún valor máximo  $C$  para el cual la intersección de la recta  $x_1 \leq x_2 \leq D$  y la región factible no es vacía. Cualquier punto en el que esto ocurra es una solución óptima del programa lineal, que en este caso es el punto  $x_1 = 2$  y  $x_2 = 6$  con valor objetivo 8.

No es casualidad que una solución óptima del programa lineal ocurra en un vértice de la región factible. El valor máximo de  $C$  para el cual la línea  $x_1 \leq x_2 \leq D$  corta la región factible debe estar en el límite de la región factible y, por lo tanto, la intersección de esta línea con el límite de la región factible es un solo vértice o una línea segmento. Si la intersección es un solo vértice, entonces solo hay una solución óptima, y es ese vértice. Si la intersección es un segmento de línea, cada punto en ese segmento de línea debe tener el mismo valor objetivo; en particular, ambos extremos del segmento de línea son soluciones óptimas. Dado que cada extremo de un segmento de línea es un vértice, también en este caso hay una solución óptima en un vértice.

Aunque no podemos graficar fácilmente programas lineales con más de dos variables, se mantiene la misma intuición. Si tenemos tres variables, entonces cada restricción corresponde a un medio espacio en el espacio tridimensional. La intersección de estos semiespacios forma la región factible. El conjunto de puntos para los que la función objetivo obtiene un valor dado  $C$  es ahora un plano (asumiendo que no hay condiciones degeneradas). Si todos los coeficientes de la función objetivo son no negativos, y si el origen es una solución factible del programa lineal, entonces a medida que alejamos este plano del origen, en una dirección normal a la función objetivo, encontramos puntos de valor objetivo creciente. (Si el origen no es factible o si algunos coeficientes en la función objetivo son negativos, la imagen intuitiva se vuelve un poco más complicada). Como en dos dimensiones, debido a que la región factible es convexa, el conjunto de puntos que logran el valor objetivo óptimo debe incluir un vértice de la región factible. De manera similar, si tenemos  $n$  variables, cada restricción define un semiespacio en un espacio  $n$ -dimensional. Llamamos simplex a la región factible formada por la intersección de estos semiespacios. La función objetivo es ahora un hiperplano y, debido a la convexidad, todavía se presenta una solución óptima en un vértice del simplex.

El algoritmo simplex toma como entrada un programa lineal y devuelve una solución óptima. Comienza en algún vértice del simplex y realiza una secuencia de iteraciones. En cada iteración, se mueve a lo largo de un borde del simplex desde un vértice actual hasta un vértice vecino cuyo valor objetivo no es menor que el del vértice actual (y generalmente es mayor). El algoritmo simplex termina cuando alcanza un máximo local, que es un vértice a partir del cual todos los vértices vecinos tienen un valor objetivo menor. Debido a que la región factible es convexa y la función objetivo es lineal, este óptimo local es en realidad un óptimo global. En la Sección 29.4,

Usaremos un concepto llamado "dualidad" para mostrar que la solución devuelta por el algoritmo simplex es realmente óptima.

Aunque la vista geométrica ofrece una buena visión intuitiva de las operaciones del algoritmo simplex, no nos referiremos a ella explícitamente cuando desarrollemos los detalles del algoritmo simplex en la sección 29.3. En su lugar, tomamos una vista algebraica. Primero escribimos el programa lineal dado en forma holgada, que es un conjunto de igualdades lineales.

Estas igualdades lineales expresan algunas de las variables, llamadas "variables básicas", en términos de otras variables, llamadas "variables no básicas". Nos movemos de un vértice a otro haciendo que una variable básica se vuelva no básica y haciendo que una variable no básica se vuelva básica. Llamamos a esta operación un "pivot" y, vista algebraicamente, no es más que reescribir el programa lineal en una forma de holgura equivalente.

El ejemplo de dos variables descrito anteriormente fue particularmente simple. Tendremos que abordar varios detalles más en este capítulo. Estos problemas incluyen la identificación de programas lineales que no tienen soluciones, programas lineales que no tienen una solución óptima finita y programas lineales para los que el origen no es una solución factible.

### Aplicaciones de la programación lineal

La programación lineal tiene un gran número de aplicaciones. Cualquier libro de texto sobre investigación de operaciones está repleto de ejemplos de programación lineal, y la programación lineal se ha convertido en una herramienta estándar que se enseña a los estudiantes en la mayoría de las escuelas de negocios. El escenario electoral es un ejemplo típico. Dos ejemplos más de programación lineal son los siguientes:

Una aerolínea desea programar sus tripulaciones de vuelo. La Administración Federal de Aviación impone muchas restricciones, como limitar el número de horas consecutivas que cada miembro de la tripulación puede trabajar e insistir en que una tripulación en particular trabaje solo en un modelo de aeronave durante cada mes. La aerolínea quiere programar tripulaciones en todos sus vuelos utilizando la menor cantidad posible de tripulantes.

Una compañía petrolera quiere decidir dónde perforar en busca de petróleo. Ubicar un taladro en un lugar particular tiene un costo asociado y, con base en estudios geológicos, un pago esperado de cierta cantidad de barriles de petróleo. La empresa tiene un presupuesto limitado para ubicar nuevos taladros y quiere maximizar la cantidad de petróleo que espera encontrar, dado este presupuesto.

Con los programas lineales, también modelamos y resolvemos problemas gráficos y combinatorios, como los que aparecen en este libro de texto. Ya vimos un caso especial de programación lineal utilizada para resolver sistemas de restricciones en diferencias en la sección 24.4. En la Sección 29.2, estudiaremos cómo formular varios problemas de flujo de redes y gráficos como programas lineales. En la Sección 35.4, usaremos la programación lineal como una herramienta para encontrar una solución aproximada a otro problema gráfico.

### Algoritmos para programación lineal

Este capítulo estudia el algoritmo simplex. Este algoritmo, cuando se implementa con cuidado, a menudo resuelve rápidamente los programas lineales generales en la práctica. Sin embargo, con algunas entradas cuidadosamente diseñadas, el algoritmo simplex puede requerir un tiempo exponencial. El primer algoritmo de tiempo polinomial para la programación lineal fue el algoritmo elíptico, que en la práctica funciona lentamente. Una segunda clase de algoritmos de tiempo polinomial se conocen como métodos de punto interior. A diferencia del algoritmo simplex, que se mueve por el exterior de la región factible y mantiene una solución factible que es un vértice del simplex en cada iteración, estos algoritmos se mueven por el interior de la región factible. Las soluciones intermedias, aunque factibles, no son necesariamente vértices del simplex, pero la solución final es un vértice. Para entradas grandes, los algoritmos de punto interior pueden ejecutarse tan rápido y, a veces, más rápido que el algoritmo simplex. Las notas del capítulo le indican más información sobre estos algoritmos.

Si a un programa lineal le sumamos el requisito adicional de que todas las variables tomen valores enteros, tenemos un programa lineal entero. El ejercicio 34.5-3 le pide que demuestre que encontrar una solución factible a este problema es NP-difícil; dado que no se conocen algoritmos de tiempo polinomial para ningún problema NP-difícil, no existe un algoritmo de tiempo polinomial conocido para la programación lineal entera. Por el contrario, podemos resolver un problema general de programación lineal en tiempo polinomial.

En este capítulo, si tenemos un programa lineal con variables  $x \in D = \{x_1; x_2; \dots; x_n\}$  y deseamos referirnos a un ajuste particular de las variables, usaremos la notación  $x \in N \subseteq D = \{x_{N1}; x_{N2}; \dots; x_{Nn}\}$ .

## 29.1 Formularios estándar y de holgura

Esta sección describe dos formatos, la forma estándar y la forma holgada, que son útiles cuando especificamos y trabajamos con programas lineales. En forma estándar, todas las restricciones son desigualdades, mientras que en forma flexible, todas las restricciones son igualdades (excepto aquellas que requieren que las variables no sean negativas).

### Forma estándar

En forma estándar, tenemos  $n$  números reales  $c_1; c_2; \dots; c_n$ ;  $m$  números reales  $b_1; b_2; \dots; b_m$ ; y  $mn$  números reales  $a_{ij}$  para  $i \in \{1; 2; \dots; m\}$  y  $j \in \{1; 2; \dots; n\}$ . Deseamos encontrar  $n$  números reales  $x_1; x_2; \dots; x_n$  que

norte.

$$\text{maximizar } \sum_{j=1}^n c_j x_j \quad (29.16)$$

sujeto a

$$\sum_{j=1}^n a_{ij} x_j \leq b_i \text{ para } i = 1, 2, \dots, m \quad (29.17)$$

$$x_j \geq 0 \text{ para } j = 1, 2, \dots, n \quad (29.18)$$

Generalizando la terminología que presentamos para el programa lineal de dos variables, llamamos a la expresión (29.16) la función objetivo ya las  $n$  desigualdades en las líneas (29.17) y (29.18) las restricciones. Las  $n$  restricciones en la línea (29.18) son las restricciones de no negatividad. Un programa lineal arbitrario no necesita tener restricciones de no negatividad, pero la forma estándar las requiere. A veces encontramos conveniente expresar un programa lineal en una forma más compacta. Si creamos una matriz  $m \times n$   $A$ , un vector  $m$   $b$ , un vector  $n$   $c$  y un vector  $n$   $x$ , entonces podemos reescribir el programa lineal definido en (29.16)–(29.18) como

$$\text{maximizar } c^T x \quad (29.19)$$

sujeto a

$$Ax \leq b \quad (29.20)$$

$$x \geq 0 \quad (29.21)$$

En la línea (29.19),  $c^T x$  es el producto interno de dos vectores. En la desigualdad (29.20),  $Ax$  es un producto matriz-vector, y en la desigualdad (29.21),  $x \geq 0$  significa que cada entrada del vector  $x$  debe ser no negativa. Vemos que podemos especificar un programa lineal en forma estándar mediante una tupla  $(A; b; c)$ , y adoptaremos la convención de que  $A$ ,  $b$  y  $c$  siempre tienen las dimensiones dadas arriba.

Ahora presentamos terminología para describir soluciones a programas lineales. Usamos parte de esta terminología en el ejemplo anterior de un programa lineal de dos variables. Llamamos solución factible a un ajuste de las variables  $x_N$  que satisface todas las restricciones, mientras que un ajuste de las variables  $x_N$  que no satisface al menos una restricción es una solución no factible. Decimos que una solución  $x_N$  tiene valor objetivo  $c^T x_N$ . Una solución factible  $x_N$  cuyo valor objetivo es máximo sobre todas las soluciones factibles es una solución óptima, y llamamos a su valor objetivo  $c^T x_N$  el valor objetivo óptimo. Si un programa lineal no tiene soluciones factibles, decimos que el programa lineal es in factible; de lo contrario es factible. Si un programa lineal tiene algunas soluciones factibles pero no tiene un valor objetivo óptimo finito, decimos que el programa lineal es ilimitado. El ejercicio 29.1-9 le pide que demuestre que un programa lineal puede tener un valor objetivo óptimo finito incluso si la región factible no está acotada.

### Conversión de programas lineales a formato estándar

Siempre es posible convertir un programa lineal, dado que minimiza o maximiza una función lineal sujeta a restricciones lineales, en forma estándar. Un programa lineal podría no estar en forma estándar por cualquiera de las siguientes cuatro posibles razones:

1. La función objetivo podría ser una minimización en lugar de una maximización.
2. Puede haber variables sin restricciones de no negatividad.
3. Puede haber restricciones de igualdad, que tienen un signo igual en lugar de un signo menor o igual que.
4. Puede haber restricciones de desigualdad, pero en lugar de tener un menor que o signo igual a, tienen un signo mayor que o igual a.

Al convertir un programa lineal  $L$  en otro programa lineal  $L_0$  como la \_\_\_\_\_, lo haríamos propiedad de que una solución óptima para  $L_0$  produce una solución óptima para  $L$ . Para capturar esta idea, decimos que dos programas lineales de maximización  $L$  y  $L_0$  son equivalentes si para cada solución factible  $x_N$  a  $L$  con valor objetivo ' $\cdot$ ', existe una solución factible correspondiente  $x_{N0}$  a  $L_0$  con valor objetivo ' $\cdot$ ', y para cada solución factible  $x_{N0}$  a  $L_0$  con valor objetivo ' $\cdot$ ', existe una solución factible correspondiente  $x_N$  a  $L$  con valor objetivo ' $\cdot$ '. (Esta definición no implica una correspondencia biunívoca entre soluciones factibles). Un programa lineal de minimización  $L$  y un programa lineal de maximización  $L_0$  son equivalentes si para cada solución factible  $x_N$  a  $L$  con valor objetivo ' $\cdot$ ', existe una solución factible correspondiente  $x_{N0}$  a  $L_0$  con valor objetivo ' $\cdot$ ', y para cada solución factible  $x_{N0}$  a  $L_0$  con valor objetivo ' $\cdot$ ', existe una correspondiente solución factible  $x_N$  a  $L$  con valor objetivo ' $\cdot$ '.

Ahora mostramos cómo eliminar, uno por uno, cada uno de los posibles problemas de la lista anterior. Después de eliminar cada uno, argumentaremos que el nuevo programa lineal es equivalente al anterior.

Para convertir un programa lineal de minimización  $L$  en un programa lineal de maximización equivalente  $L_0$ , simplemente negamos los coeficientes en la función objetivo. Como  $L$  y  $L_0$  tienen conjuntos idénticos de soluciones factibles y, para cualquier solución factible, el valor objetivo en  $L$  es el negativo del valor objetivo en  $L_0$ , estos dos programas lineales son equivalentes. Por ejemplo, si tenemos el programa lineal

$$\begin{array}{ll} \text{minimizar} & 2x_1 + 3x_2 \\ \text{sujeto a} & \begin{aligned} x_1 + x_2 &\leq 7 \\ x_1 &\leq 4 \\ x_1 &\geq 0 \end{aligned} \end{array}$$

y negamos los coeficientes de la función objetivo, obtenemos

$$\begin{array}{ll}
 \text{maximizar} & 2x_1 \quad 3x_2 \\
 \text{sujeto a} & \\
 & x_1 \leq C \quad x_2 \leq D \quad 2x_2 \\
 & x_1 \leq 4 \\
 & x_1 \geq 0
 \end{array}$$

A continuación, mostramos cómo convertir un programa lineal en el que algunas de las variables no tienen restricciones de no negatividad en uno en el que cada variable tiene una restricción de no negatividad. Suponga que alguna variable  $x_j$  no tiene una restricción de no negatividad. Luego, reemplazamos cada aparición de  $x_j$  por  $x_0 - x_{0j}$  y agregamos las restricciones de negatividad  $x_0 - x_{0j} \geq 0$ . Así, si la función objetivo tiene  $a_j$  y si la termino  $c_j x_j$ , lo reemplazamos por  $c_j x_0 - c_j x_{0j}$ . nosotros sustituirlo por  $a_j x_0 - a_j x_{0j}$ . Cualquier solución factible  $xy$  al nuevo programa lineal cor  $y x_0$  responde a una solución factible  $xN$  del programa lineal original con  $xN_j \leq y x_0$  y con el mismo valor objetivo. Además, cualquier solución factible  $xN$  del programa lineal original corresponde a una solución factible  $xy$  del nuevo programa lineal con  $x_0 \leq N_j$  y  $x_0 \leq 0$  si  $N_j < 0$ . Los dos programas lineales tienen el mismo valor objetivo independiente del signo de  $N_j$ . Por lo tanto, los dos programas lineales son equivalentes. Aplicamos este esquema de conversión a cada variable que no tiene una restricción de no negatividad para generar un programa lineal equivalente en el que todas las variables tienen restricciones de no negatividad.

Continuando con el ejemplo, queremos asegurarnos de que cada variable tenga una restricción de no negatividad correspondiente. La variable  $x_1$  tiene esa restricción, pero la variable  $x_2$  no. Por lo tanto, reemplazamos  $x_2$  por dos variables  $x_0$  y modificamos el ~~programa~~ para obtener

$$\begin{array}{ll}
 \text{maximizar} & 2x_1 \quad 3x_0 - 2 \quad C \quad 3x_0 + 2 \\
 \text{sujeto a} & \\
 & x_1 \leq x_0 - 2 \quad x_0 \leq D \quad 7 \\
 & x_1 \leq 2x_0 \quad \text{do } 2x_0 \leq 4 \\
 & x_1; x_0 \geq 0
 \end{array} \tag{29.22}$$

A continuación, convertimos las restricciones de igualdad en restricciones de desigualdad.

Suponga que un programa lineal tiene una restricción de igualdad  $f(x_1; x_2; \dots; x_n) = b$ . Como  $x_D$  y sólo si tanto  $xy$  como  $xy$ , podemos reemplazar esta restricción de igualdad por el par de restricciones de desigualdad  $f(x_1; x_2; \dots; x_n) \leq b$  y  $f(x_1; x_2; \dots; x_n) \geq b$ .

La repetición de esta conversión para cada restricción de igualdad produce un programa lineal en el que todas las restricciones son desigualdades.

Finalmente, podemos convertir las restricciones mayor o igual que en restricciones menor o igual que multiplicando estas restricciones por 1. Es decir, cualquier desigualdad de la forma

$$\sum_{j=1}^n a_{ij} x_j \leq b_i$$

es equivalente a

$$\sum_{j=1}^n a_{ij} x_j \leq b_i : \quad \text{Bi :}$$

Por lo tanto, reemplazando cada coeficiente  $a_{ij}$  por  $a_{ij}$  y cada valor  $b_i$  por  $b_i$ , obtenemos una restricción equivalente menor que o igual a.

Terminando nuestro ejemplo, reemplazamos la igualdad en la restricción (29.22) por dos en igualdades, obteniendo

$$\begin{array}{lll} \text{maximizar } & 2x_1 & 3x_0 \\ & & + C 3x_0 \\ \text{sujeto a} & & \\ & x_1 + x_0 = 2 & x_0 = 7 \\ & x_1 + x_0 = 2 & x_0 = 7 \\ & x_1 + 2x_0 = 4 & \\ & x_1; x_0, x_0 & 0 \end{array} \quad (29.23)$$

Finalmente, negamos la restricción (29.23). Por consistencia en los nombres de las variables, llamamos  $x_0$  a  $x_2$  y  $x_0$  a  $x_3$ , obteniendo la forma estándar

$$\text{maximizar } 2x_1 \quad 3x_2 + C 3x_3 \quad (29.24)$$

$$\begin{array}{lll} \text{sujeto a} & & \\ & x_1 + x_2 = 7 & x_3 = 7 \\ & x_1 + x_2 + x_3 = 7 & \\ & x_1 + 2x_3 = 4 & \\ & x_1; x_2; x_3 & 0 \end{array} \quad \begin{array}{l} (29.25) \\ (29.26) \\ (29.27) \\ (29.28) \end{array}$$

#### Conversión de programas lineales en forma floja

Para resolver eficientemente un programa lineal con el algoritmo simplex, preferimos expresarlo en una forma en la que algunas de las restricciones sean restricciones de igualdad. Más precisamente, lo convertiremos en una forma en la que las restricciones de no negatividad sean las únicas restricciones de desigualdad y las restricciones restantes sean igualdades. Dejar

$$\sum_{j=1}^n a_{ij} x_j \leq b_i \quad (29.29)$$

ser una restricción de desigualdad. Introducimos una nueva variable  $s$  y reescribimos la desigualdad (29.29) como las dos restricciones

$$s \leq b_i - \sum_{j=1}^n a_{ij} x_j; \quad (29.30)$$

$$s \geq 0. \quad (29.31)$$

Llamamos  $s$  a variable de holgura porque mide la holgura, o diferencia, entre los lados izquierdo y derecho de la ecuación (29.29). (Pronto veremos por qué nos resulta conveniente escribir la restricción con solo la variable de holgura en el lado izquierdo). Como la desigualdad (29.29) es verdadera si y solo si tanto la ecuación (29.30) como la desigualdad (29.31) son verdaderas, podemos convertir cada restricción de desigualdad de un programa lineal de esta manera para obtener un programa lineal equivalente en el que las únicas restricciones de desigualdad son las restricciones de no negatividad. Al convertir de forma estándar a holgura, usaremos  $x_{n+i}$  (en lugar de  $s$ ) para denotar la variable de holgura asociada con la  $i$ -ésima desigualdad. Por lo tanto, la  $i$ -ésima restricción es

$$x_{n+i} \leq b_i - \sum_{j=1}^n a_{ij} x_j; \quad (29.32)$$

junto con la restricción de no negatividad  $x_{n+i} \geq 0$ .

Al convertir cada restricción de un programa lineal en forma estándar, obtenemos un programa lineal en una forma diferente. Por ejemplo, para el programa lineal descrito en (29.24)–(29.28), introducimos las variables de holgura  $x_4$ ,  $x_5$  y  $x_6$ , obteniendo

$$\text{maximizar} \quad 2x_1 \quad 3x_2 \leq 3x_3 \quad (29.33)$$

$$\text{sujeto a} \quad x_4 \leq 7 - x_5 \quad x_1 \quad x_2 \leq x_3 \quad (29.34)$$

$$7 - x_1 \leq x_2 \leq x_6 \leq 4 - x_1 \leq x_3 \quad (29.35)$$

$$2x_3 \quad (29.36)$$

$$x_1; x_2; x_3; x_4; x_5; x_6 \quad (29.37)$$

En este programa lineal, todas las restricciones excepto las restricciones de no negatividad son igualdades y cada variable está sujeta a una restricción de no negatividad. Escribimos cada restricción de igualdad con una de las variables en el lado izquierdo de la igualdad y todas las demás en el lado derecho. Además, cada ecuación tiene el mismo conjunto de variables en el lado derecho, y estas variables también son las únicas que aparecen en la función objetivo. Llamamos a las variables del lado izquierdo de las igualdades variables básicas ya las del lado derecho variables no básicas.

Para los programas lineales que satisfacen estas condiciones, a veces omitiremos las palabras "maximizar" y "sujeto a", así como las restricciones explícitas de no negatividad. También utilizaremos la variable  $\cdot$  para denotar el valor de la función objetivo.

ción Llamamos al formato resultante formulario de holgura. Si escribimos el programa lineal dado en (29.33)–(29.37) en forma holgada, obtenemos

$$\begin{array}{lll} D & 2x_1 & 3x_2 \quad C \quad 3x_3 \\ x_4 \quad D \quad 7 \quad x_5 \quad D & x_1 & x_2 \quad C \quad x_3 \\ 7 \quad C \quad x_1 \quad C \quad x_2 \quad x_6 \quad D \quad 4 \quad x_1 \quad C \quad 2x_2 & & x_3 \\ & & 2x_3 : \end{array} \quad (29.38)$$

$$(29.39)$$

$$(29.40)$$

$$(29.41)$$

Al igual que con la forma estándar, nos parece conveniente tener una notación más concisa para describir una forma floja. Como veremos en la sección 29.3, los conjuntos de variables básicas y no básicas cambiarán a medida que se ejecute el algoritmo simplex. Usamos N para denotar el conjunto de índices de las variables no básicas y B para denotar el conjunto de índices de las variables básicas. Siempre tenemos que  $j \in N$ ,  $j \notin B$ ,  $i \in B$ , y  $N \cup B = \{1, 2, \dots, n\}$ . Las ecuaciones están indexadas por las entradas de B, y las variables del lado derecho están indexadas por las entradas de N. Usamos  $b_{ij}$  para denotar términos y coeficientes constantes. También usamos  $c_j$  para denotar un término constante opcional en la función objetivo. (Veremos un poco más adelante que la inclusión del término constante en la función objetivo facilita la determinación del valor de la función objetivo). Así, podemos definir de manera concisa una forma holgada mediante una tupla  $(N; B; A; b; C; c)$ , que denota la forma floja

$$\begin{array}{lll} D & CX & c_j x_j \\ & j \in N & \\ & & \vdots \\ & & j \in N \end{array} \quad (29.42)$$

$$\begin{array}{lll} x_i \quad D \quad b_i \quad X \quad a_{ij} \quad x_j \quad \text{para } i \in B \\ & j \in N & \\ & & \vdots \\ & & j \in N \end{array} \quad (29.43)$$

en el que todas las variables  $x$  están restringidas a ser no negativas. Debido a que restamos la suma  $\sum_{j \in N} a_{ij} x_j$  en (29.43), los valores  $a_{ij}$  son en realidad los negativos de los coeficientes tal como "aparecen" en la forma de holgura.

Por ejemplo, en la forma floja

$$\begin{array}{llll} D & 2x_1 & x_3 & x_5 & 2x_6 \\ x_1 \quad D \quad 8 \quad C & \frac{6x_3}{2} - & C & \frac{6x_5}{3} - & \frac{3x_6}{3} \\ x_2 \quad D \quad 4 & \frac{6}{2}x_3 & \frac{6}{2}x_5 & C & \frac{x_6}{3} \\ x_4 \quad D \quad 18 & \frac{3x_3}{2} - & C & \frac{3x_5}{2} - & \vdots \end{array}$$

tenemos  $D = f_1; 2; 4g$ ,  $S = f_3; 5; 6g$ ,

	a13 a15 a16	1=6 1=6 1=3 8=3 2=3
ANUNCIO	a23 a25 a26	D 1=3 1=2 1=2 0
	a43 a45 a46	;

	b1	;
b D	b2	D 8
	b4	4 18

c re c3 c5 c6      <sup>T</sup> D 1=6 1=6 2=3 y D 28. Tenga en cuenta que los índices de A, b y c no son necesariamente conjuntos de números enteros contiguos; dependen de los conjuntos de índices B y N .      Como ejemplo de que las entradas de A son los negativos de los coeficientes tal como aparecen en la forma holgada, observe que la ecuación para  $x_1$  incluye el término  $x_3=6$ , pero el coeficiente  $a_{13}$  es en realidad 1=6 en lugar de C1=6.

### Ejercicios

#### 29.1-1

Si expresamos el programa lineal en (29.24)–(29.28) en la notación compacta de (29.19)–(29.21), ¿qué son n, m, A, b y c?

#### 29.1-2

Dé tres soluciones factibles al programa lineal en (29.24)–(29.28). ¿Cuál es el valor objetivo de cada uno?

#### 29.1-3

Para la forma de holgura en (29.38)–(29.41), ¿cuáles son N , B, A, b, c y ?

#### 29.1-4

Convierta el siguiente programa lineal a la forma estándar:

minimizar  $2x_1 + 7x_2 + 3x_3$  sujeto a

x1	x3 D 7
3x1 + Cx2 –	24
x2	0
x3	0 ;

## 29.1-5

Convierta el siguiente programa lineal en forma holgada:

maximizar	$2x_1$	$6x_3$
sujeto a		
	$x_1 \leq x_2$	$x_3 \leq 7$
	$3x_1 \leq x_2$	$x_3 \leq 8$
	$x_1 \leq 2x_2 \leq 2x_3$	$x_3 \leq 0$
	$x_1; x_2; x_3$	$x_3 \leq 0$

¿Cuáles son las variables básicas y no básicas?

## 29.1-6

Demuestre que el siguiente programa lineal no es factible:

maximizar	$3x_1$	$2x_2$
sujeto a		
	$x_1 \leq x_2$	$2 \leq$
	$2x_1 \leq 2x_2$	$10 \leq$
	$x_1; x_2$	$0 \leq$

## 29.1-7

Muestre que el siguiente programa lineal es ilimitado:

maximizar	$x_1$	$x_2$
sujeto a		
	$2x_1 \leq x_2 \leq 2x_2$	$1 \leq$
	$x_1$	$2 \leq$
	$x_1; x_2$	$0 \leq$

## 29.1-8

Supongamos que tenemos un programa lineal general con  $n$  variables y  $m$  restricciones, y supongamos que lo convertimos a su forma estándar. Proporcione un límite superior para el número de variables y restricciones en el programa lineal resultante.

## 29.1-9

Dé un ejemplo de un programa lineal para el cual la región factible no está acotada, pero el valor objetivo óptimo es finito.

## 29.2 Formulación de problemas como programas lineales

Aunque en este capítulo nos concentraremos en el algoritmo simplex, también es importante poder reconocer cuándo podemos formular un problema como un programa lineal.

Una vez que presentamos un problema como un programa lineal de tamaño polinomial, podemos resolverlo en tiempo polinomial mediante el algoritmo del elipsoide o métodos de punto interior. Varios paquetes de software de programación lineal pueden resolver problemas de manera eficiente, de modo que una vez que el problema tiene la forma de un programa lineal, dicho paquete puede resolverlo.

Veremos varios ejemplos concretos de problemas de programación lineal. Comenzamos con dos problemas que ya hemos estudiado: el problema de los caminos más cortos de fuente única (consulte el Capítulo 24) y el problema del flujo máximo (consulte el Capítulo 26).

Luego describimos el problema de flujo de costo mínimo. Aunque el problema de flujo de costo mínimo tiene un algoritmo de tiempo polinomial que no se basa en la programación lineal, no describiremos el algoritmo. Finalmente, describimos el problema del flujo de múltiples productos básicos, para el cual el único algoritmo de tiempo polinomial conocido se basa en la programación lineal.

Cuando resolvemos problemas de gráficas en la Parte VI, usamos notación de atributos, como :d y .u; //f. Sin embargo, los programas lineales suelen utilizar variables con subíndices en lugar de objetos con atributos adjuntos. Por tanto, cuando expresemos variables en programas lineales, indicaremos los vértices y las aristas mediante subíndices.

Por ejemplo, denotamos el peso del camino más corto para el vértice no por :d sino por d.

De manera similar, denotamos el flujo desde el vértice u al vértice no por .u; //f pero por fu.

Para cantidades que se dan como entradas a problemas, como pesos de borde o capacidades, continuaremos usando notaciones como wu; / y cu:/.

### Caminos más cortos

Podemos formular el problema de los caminos más cortos de fuente única como un programa lineal.

En esta sección, nos centraremos en cómo formular el problema de la ruta más corta de un solo par, dejando la extensión al problema más general de las rutas más cortas de una sola fuente como en el ejercicio 29.2-3.

En el problema de la ruta más corta de un solo par, tenemos un gráfico dirigido y ponderado  $G = (V, E)$ , con función de peso  $w: E \rightarrow \mathbb{R}$  asignando bordes a pesos de valor real, un vértice de origen  $s$  y un vértice de destino  $t$ . Deseamos calcular el valor  $d_{st}$ , que es el peso de un camino más corto de  $s$  a  $t$ . Para expresar este problema como un programa lineal, necesitamos determinar un conjunto de variables y restricciones que definen cuándo tenemos el camino más corto de  $s$  a  $t$ . Afortunadamente, el algoritmo de Bellman-Ford hace exactamente esto. Cuando el algoritmo de Bellman-Ford termina, ha calculado, para cada vértice, un valor  $d_v$  (usando aquí la notación de subíndices en lugar de la notación de atributos) tal que para cada arista  $e = (u, v) \in E$ , tenemos  $d_v \leq d_u + w(e)$ .

El vértice de origen recibe inicialmente un valor de  $D_0$ , que nunca cambia. Por lo tanto, obtenemos el siguiente programa lineal para calcular el peso del camino más corto de  $s$  a  $t$ :

$$\text{maximizar } dt \quad (29.44)$$

$$\text{sujeto a} \quad d \leq C_{wu} ; \text{ por cada arista } u; / 2E \leq D_0 \quad (29.45)$$

$$d \geq 0 \quad (29.46)$$

Es posible que se sorprenda de que este programa lineal maximice una función objetivo cuando se supone que debe calcular las rutas más cortas. No queremos minimizar la función objetivo, ya que establecer  $d_N = 0$  para todos los  $V$  produciría una solución óptima para el programa lineal sin resolver el problema de los caminos más cortos. Maximizamos porque una solución óptima al problema de los caminos más cortos establece cada  $d_N$  en  $\min_{W,u} C_{wu}$ ;  $2E \geq d_N$  de modo que  ~~$d_N \leq 0$~~  el valor más grande que es menor que  $C_{wu}$ ;  $/$ , igual a todos del conjunto  $d_N \leq C_{wu}$ ;  $/$ . Queremos maximizar  $d$  para todos los vértices en un camino más corto de  $s$  a  $t$  sujeto a estas restricciones en todos los vértices y maximizar  $dt$  logra este objetivo.

Este programa lineal tiene  $|V|$  variables  $d$ , una para cada vértice  $V$ . También tiene restricciones  $|E|$ : una para cada borde, además de la restricción adicional de que el peso del camino más corto del vértice fuente siempre tiene el valor 0.

### Caudal máximo

A continuación, expresamos el problema de flujo máximo como un programa lineal. Recuerde que tenemos un grafo dirigido  $G = (V, E)$  en la que cada arista  $u$ ;  $/$   $E$  tiene una capacidad no negativa  $c_u$ ;  $/$  0, y se distinguen dos vértices: una fuente  $s$  y un sumidero  $t$ . Como se define en la Sección 26.1, un flujo es una función de valor real no negativa  $f: V \rightarrow \mathbb{R}$  que satisface la restricción de capacidad y la conservación del caudal. Un caudal máximo es un caudal que satisface estas restricciones y maximiza el valor del caudal, que es el caudal total que sale de la fuente menos el caudal total que entra en la fuente. Por lo tanto, un flujo satisface restricciones lineales y el valor de un flujo es una función lineal. Recordando también que asumimos que  $c_u \leq D_0$  si  $u \in E$  y que no hay aristas antiparalelas, podemos expresar el problema de flujo máximo como un programa lineal:

$$\text{maximizar } X_{fs} - X_{ft} \quad (29.47)$$

$$\text{sujeto a} \quad \sum_{u \in V} X_{fu} = \sum_{u \in V} X_{uf} \quad (29.48)$$

$$\text{cada } u \in V \quad X_{fu} \leq c_u \quad (29.49)$$

$$X_{fu} \geq 0 \quad \text{por cada } u \in V \quad (29.50)$$

Este programa lineal tiene  $jV$  variables, correspondientes al flujo entre cada par de vértices, y tiene 2 restricciones.

Lo general, es más eficiente resolver un programa lineal de menor tamaño. El programa lineal en (29.47)–(29.50) tiene, para facilitar la notación, un flujo y una capacidad de 0 para cada par de vértices  $u; v$ ; / 62 E. Sería más eficiente reescribir el programa lineal para que tenga restricciones  $OV \leq CE$ . El ejercicio 29.2-5 le pide que hazlo

#### flujo de costo mínimo

En esta sección, hemos utilizado la programación lineal para resolver problemas para los que ya conocíamos algoritmos eficientes. De hecho, un algoritmo eficiente diseñado específicamente para un problema, como el algoritmo de Dijkstra para el problema de las rutas más cortas de fuente única, o el método push-relabel para el flujo máximo, a menudo será más eficiente que la programación lineal, tanto en teoría como en términos prácticos. práctica.

El verdadero poder de la programación lineal proviene de la capacidad de resolver nuevos problemas. Recuerde el problema que enfrenta el político al comienzo de este capítulo. El problema de obtener un número suficiente de votos sin gastar demasiado dinero no se resuelve con ninguno de los algoritmos que hemos estudiado en este libro, pero podemos resolverlo mediante programación lineal. Abundan los libros con problemas del mundo real que la programación lineal puede resolver. La programación lineal también es particularmente útil para resolver variantes de problemas para los que quizás no conozcamos un algoritmo eficiente.

Considere, por ejemplo, la siguiente generalización del problema de flujo máximo. Supongamos que, además de una capacidad  $c_{uv}$  por cada arista  $u; v$ , se nos da un costo de valor real  $a_{uv}$ . Como en el problema de flujo máximo, suponemos que  $c_{uv} \geq 0$  si  $u; v \in E$ , y que no hay aristas antiparalelas. Si enviamos  $f_{uv}$  unidades de flujo sobre el borde  $u; v$ , incurrimos en un costo de  $a_{uv} f_{uv}$ . También se nos da una demanda de flujo  $d_v$ . Deseamos enviar  $d_v$  unidades de flujo de  $s$  a  $v$  mientras minimizamos el costo  $\sum_{u;v \in E} a_{uv} f_{uv}$ . Este costo total  $P = \sum_{u;v \in E} a_{uv} f_{uv}$  problema se conoce como el problema de flujo de costo mínimo.

La figura 29.3(a) muestra un ejemplo del problema de flujo de costo mínimo. Deseamos enviar 4 unidades de flujo de  $s$  a  $t$  incurriendo en el costo total mínimo. Cualquier flujo legal particular, es decir, una función  $f$  que satisface las restricciones (29.48)–(29.49), incurre en un costo total de  $P = \sum_{u;v \in E} a_{uv} f_{uv}$ . Deseamos encontrar el flujo particular de 4 unidades que minimice este costo. La figura 29.3(b) muestra una solución óptima, con costo total  $P = 10$ .

Hay algoritmos de tiempo polinomial diseñados específicamente para el problema de flujo de costo mínimo, pero están más allá del alcance de este libro. Sin embargo, podemos expresar el problema de flujo de costo mínimo como un programa lineal. El programa lineal se ve similar al del problema de flujo máximo con el control adicional.

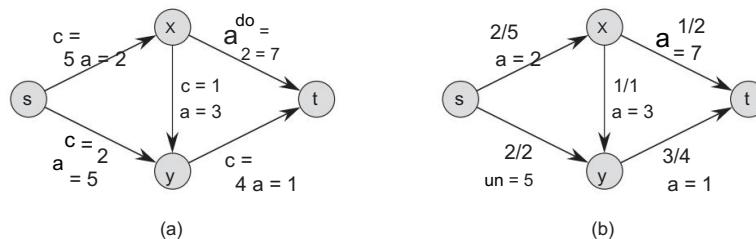


Figura 29.3 (a) Un ejemplo de un problema de flujo de costo mínimo. Denotamos las capacidades por  $c$  y los costos por  $a$ . El vértice  $s$  es la fuente y el vértice  $t$  es el sumidero, y deseamos enviar 4 unidades de flujo de  $s$  a  $t$ . (b) Una solución al problema de flujo de costo mínimo en el que se envían 4 unidades de flujo de  $s$  a  $t$ . Para cada borde, el flujo y la capacidad se escriben como flujo/capacidad.

tensión que el valor del flujo sea exactamente  $d$  unidades, y con la nueva función objetivo de minimizar el costo:

$$\begin{array}{ll} \text{minimizar } X & \text{au; /fu} \\ \text{sujeto a} & .u;/2E \end{array} \quad (29.51)$$

$X_f u$ $X_f u$ $D_0$ $2V$	$X_f s$ $X_f s$ $D_d$ $2V$	$F_u$ $cu; / por cada u; 2 voltios$ $:$	$para cada u 2 V$ $f_s; tg$ $:$	
				$(29.52)$

## Flujo de productos múltiples

Como ejemplo final, consideramos otro problema de flujo. Suponga que la empresa Lucky Puck de la Sección 26.1 decide diversificar su línea de productos y enviar no solo discos de hockey, sino también palos de hockey y cascos de hockey. Cada equipo se fabrica en su propia fábrica, tiene su propio almacén y debe enviarse, todos los días, de fábrica en almacén. Los palos se fabrican en Vancouver y deben enviarse a Saskatoon, y los cascos se fabrican en Edmonton y deben enviarse a Regina. Sin embargo, la capacidad de la red de envío no cambia y los diferentes artículos o mercancías deben compartir la misma red.

Este ejemplo es una instancia de un problema de flujo de múltiples productos. En este problema, nuevamente se nos da un grafo dirigido  $G = \langle V, E \rangle$  en la que cada arista  $e \in E$  tiene una capacidad no negativa  $c_e \geq 0$ . Como en el problema de flujo máximo, suponemos implícitamente que  $c_e = 0$  para  $e \notin E$ , y que el gráfico no tiene antipar-

aristas alélicas. Además, se nos dan  $k$  mercancías diferentes,  $K_1; K_2; \dots; K_k$ , donde especificamos la mercancía  $i$  por el triple  $K_i D_{\cdot i} ; t_i ; d_i$ . Aquí, el vértice  $s$  es la fuente de la mercancía  $i$ , el vértice  $t_i$  es el sumidero de la mercancía  $i$ , y  $d_i$  es la demanda de la mercancía  $i$ , que es el valor de flujo deseado para la mercancía de  $s$  a  $t_i$ . Definimos un flujo para el producto  $i$ , denotado por  $f_i$ , (de modo que  $f_{iu}$  es el flujo del producto  $i$  desde el vértice  $u$  hasta el vértice  $t_i$ ) como una función de valor real que satisface las restricciones de capacidad y conservación del flujo. Ahora definimos  $f_u$ , el flujo agregado, como la suma de los diversos flujos de mercancías, de modo que  $f_u = \sum_i f_{iu}$ . El flujo agregado en el borde  $u; t_i$  no debe ser mayor que la capacidad de edge  $u; t_i$ . No estamos tratando de minimizar ninguna función objetivo en este problema; sólo necesitamos determinar si tal flujo existe. Por lo tanto, escribimos un programa lineal con una función objetivo "nula":

$$\begin{aligned}
 &\text{minimizar} && 0 \\
 &\text{sujeto a} && \\
 &&& k \\
 &&& \sum_{i=1}^k f_{iu} \leq c_u; \text{ por cada } u; 2 \text{ voltios} \\
 &&& \sum_{i=1}^k f_{iu} = 0 && \text{para cada } i; 1; 2; \dots; k \text{ y} \\
 &&& f_{iu} \leq C_{ui}; \text{ para cada } u; 2 \text{ voltios} \\
 &&& f_{iu} \geq 0 && \text{por cada } u; 2 \text{ voltios} \\
 &&& f_{iu} = 0 && \text{para cada } i; 1; 2; \dots; k
 \end{aligned}$$

El único algoritmo de tiempo polinomial conocido para este problema lo expresa como un programa lineal y luego lo resuelve con un algoritmo de programación lineal en tiempo polinomial.

### Ejercicios

#### 29.2-1

Ponga el programa lineal de camino más corto de un solo par de (29.44) a (29.46) en forma estándar.

#### 29.2-2

Escriba explícitamente el programa lineal correspondiente a encontrar el camino más corto desde el nodo  $s$  hasta el nodo  $y$  en la figura 24.2(a).

#### 29.2-3

En el problema de los caminos más cortos de fuente única, queremos encontrar los pesos de los caminos más cortos desde un vértice fuente  $s$  hasta todos los vértices  $2V$ . Dada una gráfica  $G$ , escribe un

programa lineal para el cual la solución tiene la propiedad de que d es el peso del camino más corto de s a para cada vértice  $V$ .

#### 29.2-4

Escriba explícitamente el programa lineal correspondiente a encontrar el caudal máximo en la figura 26.1(a).

#### 29.2-5

Vuelva a escribir el programa lineal para el flujo máximo (29.47)–(29.50) para que use solo restricciones OV CE/.

#### 29.2-6

Escriba un programa lineal que, dado un grafo bipartito  $G = (V; E)$ , resuelve el problema de emparejamiento bipartito máximo.

#### 29.2-7

En el problema de flujo de múltiples mercancías de costo mínimo, tenemos el gráfico dirigido  $G = (V; E)$  en la que cada arista  $(u; v) \in E$  tiene una capacidad no negativa  $c_{uv} \geq 0$  y un costo  $a_{uv} \geq 0$ . Como en el problema del flujo de múltiples mercancías, tenemos  $k$  mercancías diferentes,  $K_1; K_2; \dots; K_k$ , donde especificamos la mercancía  $i$  por el triple  $(K_i, D_i, s_i, t_i, c_i)$ . Definimos el flujo  $f_i$  para el producto  $i$  y el flujo agregado  $f_u$  en el borde  $(u, v) \in E$  como en el problema del flujo de múltiples mercancías. Un flujo factible es aquel en el que el flujo agregado en cada borde  $(u, v) \in E$  no es más que la capacidad de edge  $(u, v) \in E$ . El costo de un flujo es  $P(f) = \sum_{(u, v) \in E} a_{uv} f_{uv}$ , y el objetivo es encontrar el flujo de costo mínimo. Exprese este problema como un programa lineal, y el objetivo es encontrar el flujo de costo mínimo.

### 29.3 El algoritmo simplex

El algoritmo simplex es el método clásico para resolver programas lineales. A diferencia de la mayoría de los demás algoritmos de este libro, su tiempo de ejecución no es polinomial en el peor de los casos. Sin embargo, brinda información sobre los programas lineales y, a menudo, es notablemente rápido en la práctica.

Además de tener una interpretación geométrica, descrita anteriormente en este capítulo, el algoritmo simplex tiene cierta similitud con la eliminación gaussiana, discutida en la sección 28.1. La eliminación gaussiana comienza con un sistema de igualdades lineales cuya solución se desconoce. En cada iteración, reescribimos este sistema en una forma equivalente que tiene alguna estructura adicional. Después de algunas iteraciones, hemos reescrito el sistema para que la solución sea fácil de obtener. El ritmo del algoritmo simplex procede de manera similar y podemos verlo como una eliminación gaussiana de las desigualdades.

Ahora describimos la idea principal detrás de una iteración del algoritmo simplex. Asociada con cada iteración habrá una "solución básica" que podemos obtener fácilmente de la forma holgada del programa lineal: establezca cada variable no básica en 0 y calcule los valores de las variables básicas a partir de las restricciones de igualdad. Una iteración convierte una forma de holgura en una forma de holgura equivalente. El valor objetivo de la solución factible básica asociada no será menor que el de la iteración anterior y, por lo general, mayor. Para lograr este aumento en el valor objetivo, elegimos una variable no básica tal que si aumentáramos el valor de esa variable desde 0, entonces el valor objetivo también aumentaría. La cantidad en la que podemos aumentar la variable está limitada por las otras restricciones. En particular, lo elevamos hasta que alguna variable básica se convierte en 0. Luego reescribimos la forma de holgura, intercambiando los roles de esa variable básica y la variable no básica elegida. Aunque hemos usado una configuración particular de las variables para guiar el algoritmo, y la usaremos en nuestras demostraciones, el algoritmo no mantiene explícitamente esta solución. Simplemente reescribe el programa lineal hasta que una solución óptima se vuelve "obvia".

#### Un ejemplo del algoritmo simplex

Comenzamos con un ejemplo extendido. Considere el siguiente programa lineal en forma estándar:

$$\text{maximizar } 3x_1 + 2x_2 + 2x_3 \text{ sujeto a} \quad (29.53)$$

$$x_1 + 2x_2 + 3x_3 \leq 30 \quad (29.54)$$

$$5x_1 + 4x_2 + 2x_3 \leq 24 \quad (29.55)$$

$$x_1, x_2, x_3 \geq 0 \quad (29.56)$$

$$x_1, x_2, x_3 \geq 0 \quad (29.57)$$

Para usar el algoritmo simplex, debemos convertir el programa lineal en forma holgada; vimos cómo hacerlo en la Sección 29.1. Además de ser una manipulación algebraica, la holgura es un concepto algorítmico útil. Recordando de la Sección 29.1 que cada variable tiene una restricción de no negatividad correspondiente, decimos que una restricción de igualdad es estricta para una configuración particular de sus variables no básicas si hacen que la variable básica de la restricción se convierta en 0. De manera similar, una configuración de las variables no básicas que haría que una variable básica se volviera negativa viola esa restricción. Por lo tanto, las variables de holgura mantienen explícitamente qué tan lejos está cada restricción de ser estricta y, por lo tanto, ayudan a determinar cuánto podemos aumentar los valores de las variables no básicas sin violar ninguna restricción.

Asociando las variables de holgura  $x_4$ ,  $x_5$  y  $x_6$  con las desigualdades (29.54)–(29.56), respectivamente, y poniendo el programa lineal en forma holgada, obtenemos

D	3x1 C x2 C 2x3 3x3 5x3	(29.58)
x4 profundidad 30	x1 x2	(29.59)
x5 profundidad 24	2x1 2x2	(29.60)
x6 profundidad 36	4x1 x2 2x3 :	(29.61)

El sistema de restricciones (29.59)–(29.61) tiene 3 ecuaciones y 6 variables. Cualquier configuración de las variables  $x_1$ ,  $x_2$  y  $x_3$  define valores para  $x_4$ ,  $x_5$  y  $x_6$ ; por lo tanto, tenemos un número infinito de soluciones para este sistema de ecuaciones. Una solución es factible si todo  $x_1; x_2; \dots; x_6$  no son negativos y también puede haber un número infinito de soluciones factibles. El número infinito de soluciones posibles para un sistema como este será útil en demostraciones posteriores. Nos enfocamos en la solución básica: establezca todas las variables (no básicas) en el lado derecho a 0 y luego calcule los valores de las variables (básicas) en el lado izquierdo. En este ejemplo, la solución básica es  $x_{N1}; x_{N2}; \dots; x_{N6} / D .0; 0; 0; 30; 24; 36$  y tiene valor objetivo  $\acute{D} .3 0 / C .1 0 / C .2 0 / D 0$ . Observe que esta solución básica establece  $x_{Ni} \neq 0$  para cada  $i \in \{2, B\}$ . Una iteración del algoritmo simplex reescribe el conjunto de ecuaciones y la función objetivo para poner un conjunto diferente de variables en el lado derecho. Por lo tanto, una solución básica diferente está asociada con el problema reescrito.

Hacemos hincapié en que la reescritura no cambia de ninguna manera el problema de programación lineal subyacente; el problema en una iteración tiene el mismo conjunto de soluciones factibles que el problema en la iteración anterior. Sin embargo, el problema tiene una solución básica diferente a la de la iteración anterior.

Si una solución básica también es factible, la llamamos solución factible básica. A medida que ejecutamos el algoritmo simplex, la solución básica es casi siempre una solución factible básica. Veremos en la sección 29.5, sin embargo, que para las primeras iteraciones del algoritmo simplex, la solución básica podría no ser factible.

Nuestro objetivo, en cada iteración, es reformular el programa lineal para que la solución básica tenga un mayor valor objetivo. Seleccionamos una variable no básica  $x_e$  cuyo coeficiente en la función objetivo es positivo y aumentamos el valor de  $x_e$  tanto como sea posible sin violar ninguna de las restricciones. La variable  $x_e$  se vuelve básica y alguna otra variable  $x_l$  se vuelve no básica. Los valores de otras variables básicas y de la función objetivo también pueden cambiar.

Para continuar con el ejemplo, pensemos en aumentar el valor de  $x_1$ . A medida que aumentamos  $x_1$ , los valores de  $x_4$ ,  $x_5$  y  $x_6$  disminuyen. Debido a que tenemos una restricción de no negatividad para cada variable, no podemos permitir que ninguna de ellas se vuelva negativa. Si  $x_1$  aumenta por encima de 30, entonces  $x_4$  se vuelve negativo, y  $x_5$  y  $x_6$  se vuelven negativos cuando  $x_1$  aumenta por encima de 12 y 9, respectivamente. La tercera restricción (29.61) es la restricción más estricta y limita cuánto podemos aumentar  $x_1$ . Por lo tanto, intercambiamos los roles de  $x_1$  y  $x_6$ . Resolvemos la ecuación (29.61) para  $x_1$  y obtenemos

$$x_1 D 9 \quad \frac{x_2}{4} \quad \frac{x_3}{2} \quad \frac{x_6}{4} \quad \quad \quad (29.62)$$

Para reescribir las otras ecuaciones con  $x_6$  en el lado derecho, sustituimos  $x_1$  usando la ecuación (29.62). Haciendo esto para la ecuación (29.59), obtenemos

$$x_4 D 30 x_1 x_2 3x_3 x_2 x_3$$

$$\begin{array}{rccccc} D 30 & 9 & - & \frac{x_6}{2} & & x_2 3x_3 \\ & & 2 & 4 & & \\ D 21 & \frac{3x_2}{4} & \frac{4 5x_3 x_6}{2} C & \frac{1}{4} & & \end{array} \quad (29.63)$$

De manera similar, combinamos la ecuación (29.62) con la restricción (29.60) y con la función objetivo (29.58) para reescribir nuestro programa lineal de la siguiente forma:

$$\begin{array}{rccccc} ' D 27 C & \frac{x_2}{4x_2} & C & \frac{x_3}{2x_3} & \frac{3x_6}{4x_6} & \\ & 1 & & 1 & 1 & \end{array} \quad (29.64)$$

$$\begin{array}{rccccc} x_1 D 9 & \frac{1}{4x_2} & & \frac{1}{2x_3} & \frac{1}{4x_6} & \\ & 1 & & 1 & 1 & \end{array} \quad (29.65)$$

$$\begin{array}{rccccc} x_4 D 21 & \frac{4 3x_2}{2} & & \frac{2 5x_3}{2} & C & \frac{x_6}{4} \\ & 2 & & 2 & & 1 \\ & 1 & & 1 & & 1 \end{array} \quad (29.66)$$

$$\begin{array}{rccccc} x_5 D 6 & \frac{4 3x_2}{2} & & 4x_3 C & \frac{x_6}{2} & \\ & 2 & & 1 & & 1 \\ & 1 & & 1 & & 1 \end{array} \quad (29.67)$$

Llamamos a esta operación un pivote. Como se demostró anteriormente, un pivote elige una variable no básica  $x_e$ , llamada variable de entrada, y una variable básica  $x_l$ , llamada variable de salida, e intercambia sus roles.

El programa lineal descrito en las ecuaciones (29.64)–(29.67) es equivalente al programa lineal descrito en las ecuaciones (29.58)–(29.61). Realizamos dos operaciones en el algoritmo simplex: reescribir las ecuaciones para que las variables se muevan entre el lado izquierdo y el lado derecho, y sustituir una ecuación por otra. La primera operación crea trivialmente un problema equivalente y la segunda, mediante álgebra lineal elemental, también crea un problema equivalente. (Consulte el ejercicio 29.3-3.)

Para demostrar esta equivalencia, observe que nuestra solución básica original  $.0; 0; 0; 30; 24; 36/$  satisface las nuevas ecuaciones (29.65)–(29.67) y tiene valor objetivo  $27 C .1=4/0 C .1=2/0 .3=4/36 D 0$ . La solución básica asociada con el nuevo programa lineal establece los valores no básicos en 0 y es  $.9; 0; 0; 21; 6; 0/$ , con valor objetivo  $' D 27$ . La aritmética simple verifica que esta solución también satisface las ecuaciones (29.59)–(29.61) y, cuando se reemplaza en la función objetivo (29.58), tiene un valor objetivo  $.3 9/ C .1 0/ C .2 0/ D 27$ .

Continuando con el ejemplo, deseamos encontrar una nueva variable cuyo valor deseamos aumentar. No queremos aumentar  $x_6$ , ya que a medida que aumenta su valor, el valor objetivo disminuye. Podemos intentar aumentar  $x_2$  o  $x_3$ ; elegimos  $x_3$ . ¿Hasta dónde podemos aumentar  $x_3$  sin violar ninguna de las restricciones? La restricción (29.65) lo limita a 18, la restricción (29.66) lo limita a 42=5 y la restricción (29.67) lo limita a 3=2. La tercera restricción es nuevamente la más estricta y, por lo tanto, reescribimos la tercera restricción para que  $x_3$  esté en el lado izquierdo y  $x_5$  esté en el lado derecho.

lado. Luego sustituimos esta nueva ecuación,  $x_3 D = 2$   $3x_2 = 8$   $x_5 = 4$  C  $x_6 = 8$ , en las ecuaciones (29.64)–(29.66) y obtenemos el nuevo, pero equivalente, sistema

$$\begin{array}{ccccc} & \frac{111}{D} & & & \\ & C & \frac{x_2}{\text{decidido}} & \frac{x_5}{8} & \frac{11x_6}{16} \end{array} \quad (29.68)$$

$$\begin{array}{ccccc} & \frac{4}{x_1 D} & \frac{3}{3} & \frac{x_2}{16} & \frac{5x_6}{\text{decidido}} \\ & \frac{4}{4} & & \frac{16}{C} & \end{array} \quad (29.69)$$

$$\begin{array}{ccccc} & \frac{3}{x_3 D} & & \frac{x_5}{8} & \frac{x_6}{8} \\ & \frac{—}{—} & \frac{3x_2}{\text{decidido}} & \frac{C}{C} & \end{array} \quad (29.70)$$

$$\begin{array}{ccccc} & \frac{2}{x_4 D} & \frac{6}{9} & \frac{8}{3x_2} & \frac{x_6}{16} \\ & \frac{4}{4} & C & \frac{16}{5x_5} & \frac{8}{\text{decidido}} \end{array} \quad (29.71)$$

Este sistema tiene la solución básica asociada  $.33=4; 0; 0; 3=2; 69=4; 0; 0/$ , con valor objetivo  $111=4$ .

Ahora la única forma de aumentar el valor objetivo es aumentar  $x_2$ . Las tres restricciones dan cotas superiores de 132, 4 y 1, respectivamente.

(Obtenemos un límite superior de 1 de la restricción (29.71) porque, a medida que aumentamos  $x_2$ , el valor de la variable básica  $x_4$  también aumenta. Esta restricción, por lo tanto, no impone restricciones sobre cuánto podemos aumentar  $x_2$ ). Aumentamos  $x_2$  a 4, y se vuelve no básico.

Luego resolvemos la ecuación (29.70) para  $x_2$  y sustituimos en las otras ecuaciones para obtener

$$\begin{array}{ccccc} & \frac{x_3}{D = 28} & & \frac{x_5}{2x_6} & \\ & C & & & \end{array} \quad (29.72)$$

$$\begin{array}{ccccc} & \frac{6x_3}{x_1 D = 8} & \frac{—}{C} & \frac{6x_5}{—} & \frac{3x_6}{3} \\ & \frac{—}{—} & & \frac{—}{—} & \end{array} \quad (29.73)$$

$$\begin{array}{ccccc} & \frac{6}{x_2 D = 4} & \frac{8x_3}{—} & \frac{6}{2x_5} & \frac{x_6}{3} \\ & \frac{—}{—} & & \frac{—}{—} & \end{array} \quad (29.74)$$

$$\begin{array}{ccccc} & \frac{3x_3}{x_4 D = 18} & \frac{—}{2} & \frac{3x_5}{2} & \\ & C & & & \end{array} \quad (29.75)$$

En este punto, todos los coeficientes de la función objetivo son negativos. Como veremos más adelante en este capítulo, esta situación ocurre solo cuando hemos reescrito el programa lineal para que la solución básica sea una solución óptima. Así, para este problema, la solución  $.8; 4; 0; 18; 0; 0/$ , con valor objetivo 28, es óptimo. Ahora podemos volver a nuestro programa lineal original dado en (29.53)–(29.57). Las únicas variables en el programa lineal original son  $x_1$ ,  $x_2$  y  $x_3$ , por lo que nuestra solución es  $x_1 D = 8$ ,  $x_2 D = 4$  y  $x_3 D = 0$ , con valor objetivo  $.38/C .14/C .20/D 28$ . Tenga en cuenta que los valores de las variables de holgura en la solución final miden la holgura que queda en cada desigualdad. La variable de holgura  $x_4$  es 18, y en la desigualdad (29.54), el lado izquierdo, con valor  $8C4C0D12$ , es 18 menos que el lado derecho de 30.

Las variables de holgura  $x_5$  y  $x_6$  son 0 y, de hecho, en las desigualdades (29.55) y (29.56), los lados izquierdo y derecho son iguales. Observe también que aunque los coeficientes en la forma de holgura original son enteros, los coeficientes en los otros programas lineales no son necesariamente enteros, y las soluciones intermedias no son

necesariamente integral. Además, la solución final de un programa lineal no necesita ser integral; es pura coincidencia que este ejemplo tenga una solución integral.

pivotante

Ahora formalizamos el procedimiento para pivotar. El procedimiento PIVOT toma por ejemplo una forma holgada, dada por la tupla  $.N; B; A; b; C; l;$ , el índice  $l$  de la variable de salida  $x_l$ , y el índice  $e$  de la variable de entrada  $x_e$ . Devuelve la tupla  $.N ; b; B; A;$  describiendo la nueva forma de holgura. (Recuerde nuevamente las entradas de  $c$ ; las matrices  $m n A$  y  $Ay$  son en realidad los negativos de los coeficientes que aparecen en la forma de holgura).

PIVOTE.N; B; A; b; C; ; yo; e/ 1 //

Calcular los coeficientes de la ecuación para la nueva variable básica  $x_e$ . Sea  $Ay$  una nueva matriz de  $m n 3$  y  $b$  sea  $D$  para cada

$j \leq N$  feg 5 ayej D alj  
 $=ale 6 ayel D 1=ale 7 // Calcule$   
 los coeficientes de las restricciones restantes. 8 para cada  $i \leq B$  feg  $b_i$  D bi ai eb ye 9 para cada  $j \leq N$  feg ayij D  
 $aij$  ai eayej ayi D ai eayel 12

13 // Calcule la

10 función objetivo.

11

15 DC ce ybe 14  
 for each  $j \leq N$  feg cyj D cj ceayej  
 16 17 cyl D ceayel  
 18 // Calcula nuevos conjuntos de variables básicas y no básicas.  
 19 Ny DN feg [ fgl 20 By DB fgl  
 [ feg y yb; C; A;  
 21 retorno .N ; B; /

PIVOT funciona de la siguiente manera. Las líneas 3 a 6 calculan los coeficientes en la nueva ecuación para  $x_e$  reescribiendo la ecuación que tiene  $x_l$  en el lado izquierdo para tener  $x_e$  en el lado izquierdo. Las líneas 8 a 12 actualizan las ecuaciones restantes sustituyendo el lado derecho de esta nueva ecuación por cada ocurrencia de  $x_e$ . Las líneas 14 a 17 hacen la misma sustitución de la función objetivo, y las líneas 19 y 20 actualizan la

conjuntos de variables básicas y no básicas. La línea 21 devuelve la nueva forma de holgura. Como se da, si ale D 0, PIVOT causaría un error al dividir por 0, pero como veremos en las demostraciones de los Lemas 29.2 y 29.12, llamamos PIVOT solo cuando ale  $\neq 0$ .

Ahora resumimos el efecto que PIVOT tiene sobre los valores de las variables en la solución básica.

#### Lema 29.1

Considere una llamada a PIVOT. $y_N; y_B; A; b; C; ; yo; e/$  donde ale  $\neq 0$ . Sean los valores devueltos por la llamada . $N ; b; B; A; C;$  / , y sea  $xN$  la solución básica después de la llamada. Entonces

$$1. xNj \leq 0 \text{ para cada } j \geq N . 2.$$

$$xNe = bl/ale . 3.$$

$$xNi \leq bi - ai \leq ybe \text{ para cada } i \geq 2 \text{ Por feg.}$$

Prueba La primera afirmación es verdadera porque la solución básica siempre establece todas las variables no básicas en 0. Cuando establecemos cada variable no básica en 0 en una restricción

$$\begin{aligned} xi &\leq ybi \quad \forall i \in \{1, \dots, m\} \\ &\quad j \geq N \end{aligned}$$

tenemos que  $xNi \leq ybi$  para cada  $i \geq N$ . Como  $e \geq 2$  By, la línea 3 de PIVOT da  $xNe \leq$

$ybe = bl/ale$  ; lo que

prueba la segunda afirmación. De manera similar, usando la línea 9 para cada  $i \geq 2$  Por feg, tenemos

$$xNi \leq ybi \leq bi - ai \leq ybe ; lo$$

que prueba la tercera afirmación. ■

#### El algoritmo simplex formal

Ahora estamos listos para formalizar el algoritmo simplex, que demostramos con un ejemplo. Ese ejemplo fue particularmente bueno, y podríamos haber tenido varios otros problemas que abordar:

¿Cómo determinamos si un programa lineal es factible?

¿Qué hacemos si el programa lineal es factible, pero la solución básica inicial no es factible?

¿Cómo determinamos si un programa lineal es ilimitado?

¿Cómo elegimos las variables de entrada y salida?

En la sección 29.5 mostraremos cómo determinar si un problema es factible y, de ser así, cómo encontrar una forma de holgura en la que la solución básica inicial sea factible.

Por lo tanto, supongamos que tenemos un procedimiento INITIALIZE-SIMPLEX.A; b; c/ que toma como entrada un programa lineal en forma estándar, es decir, una matriz  $m \times n$   $A$ ;  $a_{ij}$ /, un  $m$ -vector  $b$  D;  $b_i$ /, y un  $n$ -vector  $c$  D;  $c_j$ / . Si el problema no es factible, el procedimiento devuelve un mensaje de que el problema no es factible y luego termina. De lo contrario, el procedimiento devuelve una forma holgada para la cual la solución básica inicial es factible.

El procedimiento SIMPLEX toma como entrada un programa lineal en forma estándar, como se acaba de describir. Devuelve un  $n$ -vector  $x$  N D;  $x_{Nj}$ / que es una solución óptima del programa lineal descrito en (29.19)–(29.21).

SIMPLEX.A; b; C/

```

1 .N; B; A; b; C; / D INICIALIZAR-SIMPLEX.A; b; c/ 2 sea un
nuevo vector de longitud n 3 mientras que
algun índice j 2 N tiene  $c_j > 0$  4 elija un índice
e 2 N para el cual  $c_e > 0$  para cada índice i 2 B
5
6           si  $a_{ei} < 0$ 
7           i re  $b_i = a_{ei}$ 
8           demás i D 1
9           elija un índice l 2 B que minimice           i
10          si      == 1
11          devuelvo "ilimitado"
i           más .N; B; A; b; C; / D PIVOTE.N; B; A; b; C; ; yo; e/ 12 13 for
D 1 to n if i 2 B 14 15
16           $x_{Ni} = b_i$ 
else  $x_{Ni} = 0$  17
return . $x_{N1}; x_{N2}; \dots; x_{Nn}$ /
```

El procedimiento SIMPLEX funciona de la siguiente manera. En la línea 1 llama al procedimiento INITIALIZE-SIMPLEX.A; b; c/, descrito anteriormente, que determina que el programa lineal no es factible o devuelve una forma holgada para la cual la solución básica es factible. El ciclo while de las líneas 3 a 12 forma la parte principal del algoritmo. Si todos los coeficientes de la función objetivo son negativos, el ciclo while termina.

De lo contrario, la línea 4 selecciona una variable  $x_e$ , cuyo coeficiente en la función objetivo es positivo, como variable entrante. Aunque podemos elegir cualquier variable como la variable de entrada, suponemos que usamos alguna regla determinista pree especificada.

Luego, las líneas 5 a 9 verifican cada restricción y eligen la que limita más severamente la cantidad en la que podemos aumentar  $x_e$  sin violar ninguna de las restricciones no negativas.

restricciones de calidad; la variable básica asociada con esta restricción es  $x_1$ . Nuevamente, somos libres de elegir una de varias variables como la variable saliente, pero suponemos que usamos alguna regla determinista preespecificada. Si ninguna de las restricciones limita la cantidad en la que puede aumentar la variable de entrada, el algoritmo devuelve "ilimitado" en la línea 11. De lo contrario, la línea 12 intercambia los roles de las variables de entrada y salida llamando a PIVOT.N ; B; A; b; C; ; yo; e/, como se describe anteriormente.

Las líneas 13 a 16 calculan una solución  $x_{N1}$ ;  $x_{N2}$ ;:::;  $x_N$  para las variables originales de programación lineal estableciendo todas las variables no básicas en 0 y cada variable básica  $x_{Ni}$  en  $b_i$ , y la línea 17 devuelve estos valores.

Para mostrar que SIMPLEX es correcto, primero mostramos que si SIMPLEX tiene una solución factible inicial y finalmente termina, entonces devuelve una solución factible o determina que el programa lineal no tiene límites. Luego, mostramos que SIMPLEX termina. Finalmente, en la Sección 29.4 (Teorema 29.10) mostramos que la solución devuelta es óptima.

#### Lema 29.2

Dado un programa lineal .A; b; c/, suponga que la llamada a INITIALIZE-SIMPLEX en la línea 1 de SIMPLEX devuelve una forma floja para la cual la solución básica es factible.

Entonces, si SIMPLEX devuelve una solución en la línea 17, esa solución es una solución factible para el programa lineal. Si SIMPLEX devuelve "ilimitado" en la línea 11, el programa lineal es ilimitado.

Prueba Usamos el siguiente bucle invariante de tres partes:

Al comienzo de cada iteración del bucle while de las líneas 3 a 12,

1. la forma de holgura es equivalente a la forma de holgura devuelta por la llamada de INITIALIZE-SIMPLEX, 2.

para cada  $i \in B$ , tenemos  $b_i = 0$ , y

3. la solución básica asociada con la forma de holgura es factible.

Inicialización: La equivalencia de las formas flojas es trivial para la primera iteración. Suponemos, en el enunciado del lema, que la llamada a INITIALIZE-SIMPLEX en la línea 1 de SIMPLEX devuelve una forma floja para la cual la solución básica es factible. Por lo tanto, la tercera parte del invariante es verdadera. Debido a que la solución básica es factible, cada variable básica  $x_i$  es no negativa. Además, dado que la solución básica establece cada variable básica  $x_i$  en  $b_i$ , tenemos que  $b_i = 0$  para todo  $i \in B$ . Por lo tanto, la segunda parte del invariante se cumple.

Mantenimiento: mostraremos que cada iteración del ciclo while mantiene el ciclo invariable,

asumiendo que la declaración de retorno en la línea 11 no se ejecuta.

Manejaremos el caso en el que la línea 11 se ejecuta cuando discutamos la terminación.

Una iteración del ciclo while intercambia el rol de una variable básica y no básica llamando al procedimiento PIVOT . Por el Ejercicio 29.3-3, la forma de holgura es equivalente a la de la iteración anterior que, por el bucle invariante, es equivalente a la forma de holgura inicial.

Ahora demostramos la segunda parte del bucle invariante. Suponemos que al comienzo de cada iteración del bucle while ,  $b_i = 0$  para cada  $i \in B$ , y mostraremos que estas desigualdades siguen siendo verdaderas después de la llamada a PIVOT en la línea 12. Dado que los únicos cambios en las variables  $b_i$  y el conjunto  $B$  de variables básicas ocurren en esta asignación, basta con mostrar que la línea 12 mantiene esta parte del PIVOT, y  $y_{bi}$  se mantienen invariantes.  $y$   $B$  se refieren a los valores anteriores a la llamada de PIVOT .

Primero, observamos que  $b_i \neq 0$  porque  $b_i = 0$  por el bucle invariante,  $a_{ij} > 0$  por las líneas 6 y 9 de SIMPLEX, y  $b_i \neq 0$  porque  $b_i = a_{ij} / a_{ii}$  por la línea 3 de PIVOT.

Para los índices restantes  $i \in B$  fijos, tenemos que  $y_{bi} \leq D$

$a_{ij} \leq y_{bi}$  (por la línea 9 de PIVOT)

$$D \geq a_{ij} \leq y_{bi} / a_{ii} \quad (\text{por la línea 3 de PIVOT}) \quad (29.76)$$

Tenemos dos casos a considerar, dependiendo de si  $a_{ij} > 0$  o  $a_{ij} = 0$ . Si  $a_{ij} > 0$ , entonces como elegimos  $i$  tal que

$$a_{ij} = y_{bi} \quad \text{para todo } i \in B \quad ; \quad (29.77)$$

tenemos

$$y_{bi} \leq D \quad \text{para todo } i \in B \quad ; \quad (29.76)$$

$$y_{bi} \leq a_{ij} / a_{ii} \quad (\text{por desigualdad (29.77)})$$

$$y_{bi} \leq a_{ij} / a_{ii}$$

$$y_{bi} \leq D$$

y por lo tanto  $y_{bi} \leq D$ . Si  $a_{ij} = 0$ , entonces como  $a_{ij} = 0$ ,  $b_i$  y  $y_{bi}$  son no negativos, la ecuación (29.76) implica que  $y_{bi}$  también debe ser no negativo.

Ahora argumentamos que la solución básica es factible, es decir, que todas las variables tienen valores no negativos. Las variables no básicas se establecen en 0 y, por lo tanto, no son negativas. Cada variable básica  $x_i$  está definida por la ecuación

$$x_i = D - \sum_{j \in N} a_{ij} y_j$$

La solución básica establece  $x_i \geq 0$  para todos  $i \in B$ . Usando la segunda parte del invariante de ciclo, concluimos que cada variable básica  $x_i$  es no negativa.

Terminación: el ciclo while puede terminar de una de dos maneras. Si termina debido a la condición de la línea 3, entonces la solución básica actual es factible y la línea 17 devuelve esta solución. La otra forma en que termina es devolviendo "ilimitado" en la línea 11. En este caso, para cada iteración del ciclo for en las líneas 5–8, cuando se ejecuta la línea 6, encontramos que  $a_i = 0$ . Considere la solución  $x_N$  definido como

$$\begin{aligned} & \quad 1 && \text{si } y \text{ de } ; \\ x_{Ni} & D \quad 0 && \text{si } i \in N \text{ feg } ; \\ & bi p \quad j \in N \quad a_{ij}x_{Nj} && \text{si } y \text{ o } B \end{aligned}$$

Ahora mostramos que esta solución es factible, es decir, que todas las variables son no negativas. Las variables no básicas distintas de  $x_{Ne}$  son 0 y  $x_{Ne} D 1 > 0$ ; por tanto, todas las variables no básicas son no negativas. Para cada variable básica  $x_{Ni}$ , tenemos

$$\begin{aligned} x_{Ni} & D bi X a_{ij} x_{Nj} \\ & \quad j \in N \\ & re bi \quad ai ex_{Ne} : \end{aligned}$$

El bucle invariante implica que  $bi = 0$ , y tenemos  $ai \neq 0$ . Por lo tanto,  $x_{Ni} = 0$ .

Ahora mostramos que el valor objetivo para la solución  $x_N$  no está acotado. De la ecuación (29.42), el valor objetivo es

$$\begin{aligned} & D CX c_{jN} \\ & \quad j \in N \\ DC & cex Ne : \end{aligned}$$

Dado que  $c_{ej} > 0$  (por la línea 4 de SIMPLEX) y  $x_{Ne} D 1$ , el valor objetivo es 1 y, por lo tanto, el programa lineal es ilimitado. ■

Queda por demostrar que SIMPLEX termina, y cuando termina, la solución que devuelve es óptima. La sección 29.4 abordará la optimización. Ahora discutimos la terminación.

### Terminación

En el ejemplo dado al comienzo de esta sección, cada iteración del algoritmo simplex aumentó el valor objetivo asociado con la solución básica. Como le pide que muestre el ejercicio 29.3-2, ninguna iteración de SIMPLEX puede disminuir el valor objetivo asociado con la solución básica. Desafortunadamente, es posible que una iteración deje el valor objetivo sin cambios. Este fenómeno se llama degeneración, y ahora lo estudiaremos con mayor detalle.

La asignación en la línea 14 de PIVOT, DC ceb ye, cambia el valor objetivo. dado que SIMPLEX llama a PIVOT solo cuando  $c_e > 0$ , la única forma de que el objetivo permanecer sin cambios (es decir, D ) sea que b ye sea 0. Este valor se asigna para como ybe D bl= ale en la línea 3 de PIVOT Como siempre llamamos a PIVOT con ale  $\neq 0$ , vemos que para que ybe sea igual a 0, y por lo tanto el valor objetivo no cambie, debemos tener bl D 0.

De hecho, esta situación puede ocurrir. Consideré el programa lineal

$$\begin{array}{lll} D & x_1 & C \\ x_4 & D & 8 \\ x_5 & D & _ \end{array} \quad \begin{array}{lll} x_1 & C & x_2 \\ x_2 & & x_3 \\ & x_2 & x_3 : \end{array}$$

Supongamos que elegimos  $x_1$  como variable de entrada y  $x_4$  como variable de salida. Después de pivotar, obtenemos

$$\begin{array}{lll} 'D & 8 & x_3 \\ x_1 & D & 8 \\ x_5 & D & _ \end{array} \quad \begin{array}{lll} x_3 & _ & x_4 \\ x_2 & & x_4 \\ x_2 & & x_3 : \end{array}$$

En este punto, nuestra única opción es pivotar con  $x_3$  entrando y  $x_5$  saliendo. Dado que  $b \leq D 0$ , el valor objetivo de 8 permanece sin cambios después de pivotar:

$$\begin{array}{lll} 'D & 8 & C \\ x_2 & x_1 & D \\ x_3 & D & _ \end{array} \quad \begin{array}{lll} x_4 & x_5 \\ x_2 & x_4 \\ x_2 & x_3 : \end{array}$$

El valor objetivo no ha cambiado, pero nuestra forma de holgura sí. Afortunadamente, si pivotamos de nuevo, entrando  $x_2$  y saliendo  $x_1$ , el valor objetivo aumenta (a 16) y el algoritmo simplex puede continuar.

La degeneración puede evitar que el algoritmo simplex termine, porque puede conducir a un fenómeno conocido como ciclado: las formas de holgura en dos iteraciones diferentes de SIMPLEX son idénticas. Debido a la degeneración, SIMPLEX podría elegir una secuencia de operaciones pivot que deje el valor objetivo sin cambios pero repita una forma floja dentro de la secuencia. Dado que SIMPLEX es un algoritmo determinista, si realiza un ciclo, realizará un ciclo a través de la misma serie de formularios de holgura para siempre, sin terminar nunca.

El ciclo es la única razón por la que SIMPLEX podría no terminar. Para mostrar este hecho, primero debemos desarrollar alguna maquinaria adicional.

En cada iteración, SIMPLEX mantiene A, b, c, y además de los conjuntos N y B. Aunque necesitamos mantener explícitamente A, b, c, y para implementar el algoritmo simplex de manera eficiente, podemos arreglárnoslas sin mantener a ellos. En otras palabras, los conjuntos de variables básicas y no básicas son suficientes para determinar de forma única la forma de holgura. Antes de probar este hecho, demostremos un útil lema algebraico.

## Lema 29.3

Sea  $I$  un conjunto de índices. Para cada  $j \in I$ , sean  $j$  y  $\tilde{j}$  números reales, y sea  $x_j$  sea una variable de valor real. Sea cualquier número real. Supongamos que para cualquier configuración de  $x_j$ , tenemos

$$\sum_{j \in I} j x_j \leq C \sum_{j \in I} \tilde{j} x_j : \quad (29.78)$$

Entonces  $j \leq \tilde{j}$  para cada  $j \in I$ . y  $D = 0$ .

Prueba Dado que la ecuación (29.78) se cumple para cualquier valor de  $x_j$ , podemos usar valores particulares para sacar conclusiones acerca de  $x_j$ . Si hacemos  $x_j = 0$  para cada  $j \in I$ , de  $\tilde{j}$ , llegamos a la conclusión de que  $D = 0$ . Ahora elija un índice arbitrario  $j \in I$  y establezca  $x_j = 1$  y  $x_k = 0$  para todo  $k \neq j$ . Entonces debemos tener  $j \leq \tilde{j}$ . Como elegimos  $j$  como cualquier índice, concluimos que  $j \leq \tilde{j}$  para cada  $j \in I$ . ■

Un programa lineal particular tiene muchas formas de holgura diferentes; recuerde que cada forma de holgura tiene el mismo conjunto de soluciones factibles y óptimas que el programa lineal original. Ahora mostramos que la forma holgada de un programa lineal está determinada únicamente por el conjunto de variables básicas. Es decir, dado el conjunto de variables básicas, se asocia una forma de holgura única (conjunto único de coeficientes y lados derechos) con esas variables básicas.

## Lema 29.4

Sea  $A; b; c$  ser un programa lineal en forma estándar. Dado un conjunto  $B$  de variables básicas, la forma de holgura asociada se determina de manera única.

Demostración Asumir con el propósito de contradicción que hay dos formas de holgura diferentes con el mismo conjunto  $B$  de variables básicas. Las formas flojas también deben tener conjuntos idénticos  $N$   $f_1; f_2; \dots; f_n$   $C$  mg  $B$  de variables no básicas. Escribimos la primera forma de holgura como

$$\sum_{j \in N} c_j x_j \leq \sum_{j \in N} a_{ij} x_j \text{ para } i \in B \quad (29.79)$$

$$\sum_{j \in N} x_i \leq b_i \text{ para } i \in B \quad (29.80)$$

y el segundo como

$$\sum_{j \in N} c_0 x_j \leq \sum_{j \in N} a_{0j} x_j \text{ para } i \in B \quad (29.81)$$

$$\sum_{j \in N} x_i \leq b_0 \text{ para } i \in B \quad (29.82)$$

Considere el sistema de ecuaciones formado al restar cada ecuación en la línea (29.82) de la ecuación correspondiente en la línea (29.80). El sistema resultante es

$$0 D .bi b0 i / X .aij a0 \quad y_0/x_j \text{ para } i \in B \\ j \in N$$

o equivalente,

$$X aij xj D .bi b0 i / CX \quad a_{ij}^0 x_j \text{ para } i \in B \\ j \in N$$

Ahora, para cada  $i \in B$ , aplique el Lema 29.3 con  $j \in D a_{ij}$ ,  $\tilde{j} \in D a_0$  IDN. Como  $y_0$ ,  $D bi b0$  y  $y_0$ ,  $\tilde{j} \in D \setminus i$ , tenemos que  $aij D a_0$  tenemos que  $bi D b0$ . Así,  $y_0$  para cada  $j \in N \setminus B$ , y como  $D \setminus B$ , para las dos formas flojas, A y b son idénticas a A0

$y_0$ . Usando un argumento similar, el ejercicio 29.3-1 muestra que también debe darse el caso de que  $c D c_0$  y  $D^0$ , y por lo tanto que las formas flojas deben ser idénticas. ■

Ahora podemos demostrar que el ciclismo es la única razón posible por la que SIMPLEX podría no terminar

Lema 29.5 Si

SIMPLEX no logra terminar en como máximo  $nCm$  iteraciones, luego ciclos.

Demostración Por el Lema 29.4, el conjunto B de variables básicas determina únicamente una forma de holgura. Hay  $n C m$  variables y  $jBj D m$ , y por lo tanto, hay como máximo  $nCm$  formas de elegir B. Por lo tanto, solo hay como máximo  $nCm$  formas de holgura únicas. Por lo tanto, si SIMPLEX se ejecuta durante más de  $nCm$  iteraciones, debe realizar un ciclo. ■

El ciclismo es teóricamente posible, pero extremadamente raro. Podemos evitarlo eligiendo las variables de entrada y salida con algo más de cuidado. Una opción es perturbar ligeramente la entrada para que sea imposible tener dos soluciones con el mismo valor objetivo. Otra opción es desempatar eligiendo siempre la variable con el índice más pequeño, estrategia conocida como regla de Bland. Omitimos la prueba de que estas estrategias evitan el ciclismo.

Lema 29.6 Si

las líneas 4 y 9 de SIMPLEX siempre rompen empates eligiendo la variable con el índice más pequeño, entonces SIMPLEX debe terminar. ■

Concluimos esta sección con el siguiente lema.

**Lema 29.7**

Suponiendo que INITIALIZE-SIMPLEX devuelve una forma flexible para la cual la solución básica es factible, SIMPLEX informa que un programa lineal no está acotado o termina con una solución factible en la mayoría de las iteraciones.

Los lemas de prueba 29.2 y 29.6 muestran que si INITIALIZE-SIMPLEX devuelve una forma holgada para la cual la solución básica es factible, SIMPLEX informa que un programa lineal no está acotado o termina con una solución factible. Por el contrario positivo del Lema 29.5, si SIMPLEX termina con una solución factible, entonces termina en a lo sumo  $nCm$  iteraciones

■

**Ejercicios****29.3-1**

Complete la prueba del Lema 29.4 mostrando que debe darse el caso de que  $c \leq c_0$  y  $D \geq D_0$ .

**29.3-2**

Demuestre que la llamada a PIVOT en la línea 12 de SIMPLEX nunca disminuye el valor de  $Z$ .

**29.3-3**

Demuestre que la forma de holgura dada al procedimiento PIVOT y la forma de holgura que devuelve el procedimiento son equivalentes.

**29.3-4**

Supongamos que convertimos un programa lineal  $Ax = b$ ;  $C$  en forma estándar a forma floja.

Muestre que la solución básica es factible si y sólo si  $b_i \geq 0$  para  $i = 1, 2, \dots, m$ .

metro.

**29.3-5**

Resuelva el siguiente programa lineal usando SIMPLEX:

maximizar  $18x_1 + 12x_2$  sujeto a

$x_1$	$x_2$	20
$x_1$		12
	$x_2$	
$x_1, x_2$		0

29.3-6

Resuelva el siguiente programa lineal usando SIMPLEX:

$$\text{maximizar } 5x_1 \quad 3x_2$$

sujeto a

x1	x2	1
2x1	x2	2
x1; x2		0

29.3-7

Resuelva el siguiente programa lineal usando SIMPLEX:

$$\text{minimizar } x_1 + x_2 + x_3$$

sujeto a

2x1 + 7x2 + 3x3	20x1 + 5x2	10000
C 10x3		30000
x1; x2; x3		0

29.3-8

En la prueba del Lema 29.5, argumentamos que hay, como máximo,  $mC_n$  formas de elegir un conjunto B de variables básicas. Dé un ejemplo de un programa lineal en el que haya  $mC_n$  estrictamente menos que  $mC_n$  formas de elegir el conjunto B.

## 29.4 Dualidad

Hemos probado que, bajo ciertos supuestos, SIMPLEX termina. Sin embargo, aún no hemos demostrado que realmente encuentre una solución óptima para un programa lineal.

Para hacerlo, presentamos un poderoso concepto llamado dualidad de programación lineal.

La dualidad nos permite probar que una solución es realmente óptima. Vimos un ejemplo de dualidad en el Capítulo 26 con el Teorema 26.6, el teorema de corte mínimo de flujo máximo. Suponga que, dada una instancia de un problema de flujo máximo, encontramos un flujo  $f$  con valor  $j_f$ . ¿Cómo sabemos si  $f$  es un caudal máximo? Por el teorema de corte mínimo de flujo máximo, si podemos encontrar un corte cuyo valor también sea  $j_f$ , entonces hemos verificado que  $f$  es de hecho un flujo máximo. Esta relación proporciona un ejemplo de dualidad: dado un problema de maximización, definimos un problema de minimización relacionado de modo que los dos problemas tengan los mismos valores objetivos óptimos.

Dado un programa lineal en el que el objetivo es maximizar, describiremos cómo formular un programa lineal dual en el que el objetivo es minimizar y

cuyo valor óptimo es idéntico al del programa lineal original. Cuando nos referimos a programas lineales duales, llamamos al programa lineal original el primario.

Dado un programa lineal primal en forma estándar, como en (29.16)–(29.18), definimos el programa lineal dual como

$$\text{minimizar } \sum_{i \in D_1} b_i y_i \quad (29.83)$$

sujeto a

$$\sum_{i \in D_1} a_{ij} y_i \geq c_j \text{ para } j \in D_2; \dots \quad (29.84)$$

$$y_i \geq 0 \text{ para } i \in D_1; \dots \quad (29.85)$$

Para formar el dual, cambiamos la maximización por una minimización, intercambiamos los roles de los coeficientes en los lados derechos y la función objetivo, y reemplazamos cada menor que o igual a por un mayor que o igual.  $-a$ . Cada una de las  $m$  restricciones en el primal tiene una variable asociada  $y_i$  en el dual, y cada una de las  $n$  restricciones en el dual tiene una variable asociada  $x_j$  en el primal. Por ejemplo, considere el programa lineal dado en (29.53)–(29.57). El dual de este programa lineal es

$$\text{minimizar } 30y_1 + 24y_2 + 36y_3 \text{ sujeto a} \quad (29.86)$$

$$y_1 + 2y_2 + 4y_3 \leq 1 \quad (29.87)$$

$$y_3 + 3y_1 + 5y_2 \leq 2 \quad (29.88)$$

$$y_2 \leq 2 \quad (29.89)$$

$$y_1, y_2, y_3 \geq 0 \quad (29.90)$$

Mostraremos en el teorema 29.10 que el valor óptimo del programa lineal dual siempre es igual al valor óptimo del programa lineal primal. Además, el algoritmo simplex en realidad resuelve implícitamente tanto el programa lineal primario como el dual simultáneamente, proporcionando así una prueba de optimización.

Comenzamos demostrando la dualidad débil, que establece que cualquier solución factible del programa lineal primal tiene un valor no mayor que el de cualquier solución factible del programa lineal dual.

#### Lema 29.8 (Dualidad de programación lineal débil)

Sea  $x_N$  cualquier solución factible del programa lineal primal de (29.16)–(29.18) y sea  $y_N$  cualquier solución factible del programa lineal dual de (29.83)–(29.85). Entonces nosotros tenemos

$$\sum_{j \in D_2} c_j x_{Nj} \leq \sum_{i \in D_1} b_i y_{Ni}$$

Prueba que tenemos

$$\begin{array}{ll} X_n & c_j x_{nj} \\ jD_1 & \end{array} \quad \begin{array}{ll} X_n & X_m \\ jD_1 & a_{ij} y_{ni} \leq x_{nj} \text{ (por desigualdades (29.84))} \end{array}$$

$$\begin{array}{ll} D & X_m \\ & iD_1 \end{array} \quad \sum_{j=1}^m a_{ij} y_{ni} \leq x_{nj} !$$

$$\begin{array}{ll} X_m & b_i y_{ni} \\ iD_1 & \end{array} \quad \text{(por desigualdades (29.17))} . \quad \blacksquare$$

### Corolario 29.9

Sea  $x_N$  una solución factible de un programa lineal primal  $A; b; c$ , y sea  $y_N$  una solución factible del programa lineal dual correspondiente. Si

$$\begin{array}{ll} X_n & c_j x_{nj} \\ jD_1 & \end{array} \quad \begin{array}{ll} D & X_m \\ & iD_1 \end{array} \quad b_i y_{ni} ;$$

entonces  $x_N$  y  $y_N$  son soluciones óptimas de los programas lineales primal y dual, respectivamente.

Demostración Por el Lema 29.8, el valor objetivo de una solución factible del primal no puede exceder el de una solución factible del dual. El programa lineal primal es un problema de maximización y el dual es un problema de minimización. Por lo tanto, si las soluciones factibles  $x_N$  e  $y_N$  tienen el mismo valor objetivo, ninguna puede mejorarse. ■

Antes de demostrar que siempre hay una solución dual cuyo valor es igual al de una solución primal óptima, describimos cómo encontrar dicha solución. Cuando ejecutamos el algoritmo simplex en el programa lineal en (29.53)–(29.57), la iteración final arrojó la forma holgada (29.72)–(29.75) con el objetivo  $\rightarrow D 28 x_3 = 6$   
 $x_5 = 6, x_6 = 3, BD f_1 = 2, 4g y SD f_3 = 5, 6g$ . Como mostraremos a continuación, la solución básica asociada con la forma de holgura final es de hecho una solución óptima para el programa lineal; una solución óptima al programa lineal (29.53)–(29.57) es por lo tanto  $x_{N1}, x_{N2}, x_{N3} / D .8; 4; 0/$ , con valor objetivo  $.38/C .14/C .20/D 28$ . Como también mostramos a continuación, podemos leer una solución dual óptima: los negativos de los coeficientes de la función objetivo primal son los valores de las variables. Más precisamente, supongamos que la última forma floja del primal es

$$\begin{array}{ll} D & 0 \\ & \sum_{j=1}^n c_j x_j \end{array}$$

$$x_i D b_i \quad \sum_{j=1}^n a_{ij} x_j \text{ para } i \in B$$

Entonces, para producir una solución dual óptima, establecemos

$$\begin{array}{ll} nC_i & \text{si } nC_i / 2 N \\ yN_i D (c_0^0) & \text{de lo contrario :} \end{array} \quad (29.91)$$

Por lo tanto, una solución óptima para el programa lineal dual definido en (29.86)–(29.90) es  $yN_1 D 0$  (ya que  $nC_1 D 4 2 B$ ),  $yN_2 D c_0 D 1=6$  y  $yN_3 D c_0 D 2=3$ .<sup>6</sup>

Evaluando la función objetivo dual (29.86), obtenemos un valor objetivo de  $.30 0/ C .24 .1=6// C .36 .2=3// D 28$ , lo que confirma que el valor objetivo de la primal es de hecho igual al valor objetivo del dual. La combinación de estos cálculos con el Lema 29.8 produce una prueba de que el valor objetivo óptimo del programa lineal primal es 28. Ahora mostramos que este enfoque se aplica en general: podemos encontrar una solución óptima para el dual y simultáneamente probar que una solución para el primal es óptimo

#### Teorema 29.10 (Dualidad de programación lineal)

Supongamos que SIMPLEX devuelve valores  $xN D .xN_1; xN_2; \dots; xN_n$  para el programa lineal primario  $A; b; C$ . Sean  $N$  y  $B$  las variables básicas y no básicas para la forma de holgura final,  $c_0$  denote los coeficientes en la forma de holgura final y sea  $yN D .yN_1; yN_2; \dots; yN_m$  se definirá mediante la ecuación (29.91). Entonces  $xN$  es una solución óptima del programa lineal primal,  $yN$  es una solución óptima del programa lineal dual

$$\sum_{j=1}^n c_j xN_j D \sum_{i=1}^m yN_i D c_0 \quad \text{biy ni :} \quad (29.92)$$

Demostración Por el corolario 29.9, si podemos encontrar soluciones factibles  $xN$  e  $yN$  que satisfagan la ecuación (29.92), entonces  $xN$  e  $yN$  deben ser soluciones primal y dual óptimas. Ahora demostraremos que las soluciones  $xN$  e  $yN$  descritas en el enunciado del teorema satisfacen la ecuación (29.92).

Suponga que ejecutamos SIMPLEX en un programa lineal primario, como se indica en las líneas (29.16)–(29.18). El algoritmo procede a través de una serie de formularios de holgura hasta que termina con un formulario de holgura final con función objetivo

$$\sum_{j=1}^n c_j xN_j D c_0 \quad \text{jxj :} \quad (29.93)$$

Como SIMPLEX terminó con una solución, por la condición de la línea 3 sabemos que

$$c_0 j D 0 \text{ para todo } j \leq N \quad (29.94)$$

si definimos

$$c_0 D_0 \text{ para todo } j \geq B \quad : \quad (29.95)$$

podemos reescribir la ecuación (29.93) como

D 0 CX c0 jxj  
j2N

D 0 CX c0 jxj CX c0 jxj (porque c0  
j2N j2b) j D 0 si j 2 B)

D ° C X c0 jxj (porque N [ BD f1; 2; : : : ; n C mg) . (29.96)  
iD1

Para la solución básica  $x_N$  asociada con esta forma de holgura final,  $x_{Nj} \geq 0$  para todo  $j \neq N$ , y  $\sum_{j=1}^n x_{Nj} = D$ . Dado que todas las formas de holgura son equivalentes, si evaluamos la función objetivo original en  $x_N$ , debemos obtener el mismo valor objetivo:

$$\begin{aligned}
 & X_n \ c_j x_{nj} d_{jD1} \quad {}^0 C \begin{matrix} X \ c_0 jxNj \\ jD1 \end{matrix} \quad (29.97) \\
 & {}^D 0 \ CX \ c_0 jxNj \ CX \ c_0 jxNj \\
 & \quad \quad \quad j2N \quad \quad \quad j2b \\
 & {}^D 0 \ CX \ .c_0 \ j \quad 0/ \ CX \ .0 \ Nxj / \quad (29.98)
 \end{aligned}$$

Ahora mostraremos que  $y_N$ , definida por la ecuación (29.91), es factible para el programa lineal y que su valor objetivo  $P_m$  coincide con el dual igual a  $\sum P_{Nj} x_{Nj}$ . Ecua (29.97) dice que la primera y la última forma de holgura, evaluadas en  $x_N$ , son iguales. Más generalmente, la equivalencia de todas las formas flojas implica que para cualquier conjunto de valores  $x_D = x_1; x_2; \dots; x_n$ , tenemos

Xn cj xj d      C<sup>0</sup> X c0 j xj :  
iD1                    iD1

Por lo tanto, para cualquier conjunto particular de valores  $x_N D .x_{N1}; x_{N2}; \dots; x_{Nn}/$ , tenemos

$$\begin{aligned}
 & X_n \underset{jD1}{c_j x_{nj}} \\
 & \quad \text{---} C_m \\
 & \quad \circ C \underset{jD1}{X c_0 jxNj} \\
 & \quad \text{---} C_m \\
 & \quad 0 CX_n \underset{jD1}{c_0 jxNj} C \underset{jDnC1}{X c_0 jxNj} \\
 & \quad 0 CX_n c_0 jxNj CX_m \underset{iD1}{c_0 nCixNnCi} \\
 & \quad 0 CX_n c_0 jxNj CX_m \underset{iD1}{. Ny_i / xNnCi} \quad (\text{por las ecuaciones (29.91) y (29.95)}) \\
 & \quad 0 CX_n c_0 jxNj CX_m \underset{iD1}{. Ny_i / bi X_n} \underset{jD1}{a_{ij} x_{nj}} ! \quad (\text{por la ecuación (29.32)}) \\
 & \quad 0 CX_n c_0 jxNj XM \underset{iD1}{biyNi CX_m} \underset{iD1}{X_n} \underset{jD1}{. a_{ij} x_{nj} / y_{Ni}} \\
 & \quad 0 CX_n c_0 jxNj XM \underset{iD1}{biyNi CX_n} \underset{jD1}{XM} \underset{iD1}{. a_{ij} y_{Ni} / x_{nj}} \\
 & \quad 0 XM \underset{iD1}{biyNi ! CX_n} \underset{jD1}{c_0 CX_m} \underset{iD1}{. a_{ij} y_{ni} ! x_{nj}} ; \\
 & \text{de modo que}
 \end{aligned}$$

$$X_n \underset{jD1}{c_j x_{nj}} d \quad 0 XM \underset{iD1}{biyNi ! CX_n} \underset{jD1}{c_0 CX_m} \underset{iD1}{. a_{ij} y_{ni} ! x_{nj}} : \quad (29.99)$$

Aplicando el Lema 29.3 a la ecuación (29.99), obtenemos

$$0 XM \underset{iD1}{biyNi D 0} ; \quad (29.100)$$

$$c_0 CX_m \underset{iD1}{. a_{ij} y_{Ni} D c_j} \text{ para } j \in \{1, 2, \dots\} ; \quad (29.101)$$

Por la ecuación (29.100), tenemos  $\underset{iD1}{biyNi D} = 0$  y por tanto el valor objetivo que  $P_m$  del dual  $\underset{iD1}{PbiyNi}$  es igual al del primal (0). Queda por mostrar

que la solución  $y_N$  es factible para el problema dual. De las desigualdades (29.94) y las ecuaciones (29.95) tenemos (29.10)  $\sum_{j=1}^m a_{ij} y_{Ni} \leq b_j$  para todo  $j = 1, 2, \dots, n$ . Por lo tanto, para cualquier  $j$   $\sum_{i=1}^m a_{ij} y_{Ni} \geq 0$ .

$$\sum_{i=1}^m a_{ij} y_{Ni} \geq 0 \quad \forall j = 1, 2, \dots, m$$

$$\sum_{i=1}^m a_{ij} y_{Ni} \geq 0 \quad \forall j = 1, 2, \dots, m$$

que satisface las restricciones (29.84) del dual. Finalmente, dado que  $c_0 \geq 0$  para cada  $[B]$ , cuando establecemos  $y_N$  de acuerdo con la ecuación (29.91), tenemos que cada  $y_{Ni} \geq 0$ , por lo tanto, también se satisfacen las restricciones de no negatividad. ■

Hemos demostrado que, dado un programa lineal factible, si INITIALIZE-SIMPLEX devuelve una solución factible, y si SIMPLEX termina sin devolver "ilimitado", entonces la solución devuelta es de hecho una solución óptima. También hemos mostrado cómo construir una solución óptima para el programa lineal dual.

### Ejercicios

#### 29.4-1

Formule el dual del programa lineal dado en el ejercicio 29.3-5.

#### 29.4-2

Suponga que tenemos un programa lineal que no está en forma estándar. Podríamos producir el dual primero convirtiéndolo a la forma estándar y luego tomando el dual.

Si embargo, sería más conveniente poder producir el dual directamente.

Explique cómo podemos tomar directamente el dual de un programa lineal arbitrario.

#### 29.4-3

Escriba el dual del programa lineal de flujo máximo, como se indica en las líneas (29.47)–(29.50) en la página 860. Explique cómo interpretar esta formulación como un problema de corte mínimo.

#### 29.4-4

Escriba el dual del programa lineal de flujo de costo mínimo, como se indica en las líneas (29.51)–(29.52) en la página 862. Explique cómo interpretar este problema en términos de gráficos y flujos.

#### 29.4-5

Muestre que el dual del dual de un programa lineal es el programa lineal primario.

## 29.4-6

¿Qué resultado del Capítulo 26 se puede interpretar como una dualidad débil para el problema de flujo máximo?

## 29.5 La solución factible básica inicial

En esta sección, primero describimos cómo probar si un programa lineal es factible y, si lo es, cómo producir una forma holgada para la cual la solución básica es factible.

Concluimos demostrando el teorema fundamental de la programación lineal, que dice que el procedimiento SIMPLEX siempre produce el resultado correcto.

### Encontrar una solución inicial

En la sección 29.3 supusimos que teníamos un procedimiento INITIALIZE-SIMPLEX que determina si un programa lineal tiene soluciones factibles y, si las tiene, da una forma holgada para la cual la solución básica es factible. Aquí describimos este procedimiento.

Un programa lineal puede ser factible, pero la solución básica inicial podría no serlo. Considere, por ejemplo, el siguiente programa lineal:

$$\text{maximizar } 2x_1 \quad x_2 \quad (29.102)$$

$$\text{sujeto a} \quad \begin{array}{ccc} 2x_1 & x_2 & 2 \end{array} \quad (29.103)$$

$$\begin{array}{ccc} x_1 & 5x_2 & 4 \end{array} \quad (29.104)$$

$$\begin{array}{ccc} x_1; x_2 & & 0 \end{array} \quad (29.105)$$

Si tuviéramos que convertir este programa lineal a la forma holgada, la solución básica establecería  $x_1 = 0$  y  $x_2 = 0$ . Esta solución viola la restricción (29.104), por lo que no es una solución factible. Por lo tanto, INITIALIZE-SIMPLEX no puede simplemente devolver la forma de holgura obvia. Para determinar si un programa lineal tiene soluciones factibles, formularemos un programa lineal auxiliar. Para este programa lineal auxiliar, podemos encontrar (con un poco de trabajo) una forma holgada para la cual la solución básica es factible.

Además, la solución de este programa lineal auxiliar determina si el programa lineal inicial es factible y, de ser así, proporciona una solución factible con la que podemos inicializar SIMPLEX.

### Lema 29.11 Sea

Un programa lineal en forma estándar, dado como en (29.16)–(29.18). Sea  $x_0$  una nueva variable, y sea  $L_{aux}$  el siguiente programa lineal con  $n + 1$  variables:

$$\text{maximizar} \quad x_0 \quad (29.106)$$

sujeto a

$$\sum_{j=1}^n a_{ij} x_j \leq b_i \text{ para } i = 1, 2, \dots, m \quad (29.107)$$

$$x_j \geq 0 \text{ para } j = 0, 1, \dots, n \quad (29.108)$$

Entonces L es factible si y solo si el valor objetivo óptimo de Laux es 0.

Prueba Suponga que L tiene una solución factible  $x_N = x_{N1}, x_{N2}, \dots, x_{Nn}$ . Entonces la solución  $x_N = 0$  combinada con  $x_N$  es una solución factible de Laux con valor objetivo 0. Dado que  $x_0 = 0$  es una restricción de Laux y la función objetivo es maximizar  $x_0$ , esta solución debe ser óptima para Laux.

Por el contrario, suponga que el valor objetivo óptimo de Laux es 0. Entonces  $x_N = 0$ , y los valores de solución restantes de  $x_N$  satisfacen las restricciones de L. ■

Ahora describimos nuestra estrategia para encontrar una solución factible básica inicial para un programa lineal L en forma estándar:

INICIALIZAR-SIMPLEX.A; b; c / 1 sea k

el índice del mínimo  $b_i / b_k < 0$  //  $i$  es factible la  
solución básica inicial? devolver .f1; 2; : : : ; ng; fn C 1; n C 2; : : : ; nCmg; A; b; C; 0/ 3 4 formar  
Laux sumando  $x_0$  al lado izquierdo de cada restricción y estableciendo la función  
objetivo en  $x_0 = 0$  sea  $.N; B; A; b; C$  / sea la forma de holgura resultante para Laux 6 I D n C  
k 7 // Laux tiene n C 1 variables no básicas y m  
variables básicas.

8 .N; B; A; b; C / D PIVOTE.N; B; A; b; C ; yo; 0/ 9 // La solución  
básica ahora es factible para Laux. 10 iterar el bucle while de las  
líneas 3–12 de SIMPLEX hasta encontrar una solución óptima para Laux

11 si la solución óptima de Laux establece  $x_N = 0$  12 si  $x_N$   
es básico

13 realice un pivote (degenerado) para convertirlo en no básico de la  
14 forma holgada final de Laux, elimine  $x_0$  de las restricciones y restaure la función objetivo  
original de L, pero reemplace cada variable básica en esta función objetivo por el lado  
derecho de su asociado restricción

15 devolver el formulario de holgura final modificado  
16 más devuelven "no factible"

INITIALIZE-SIMPLEX funciona de la siguiente manera. En las líneas 1 a 3, probamos implícitamente la solución básica de la forma de holgura inicial para L dada por ND f1; 2; : : : ; ng, BD fn C 1; n C 2; : : : ; n C mg, xNi D bi para todo i 2 B, y xNj D 0 para todo j 2 N . (La creación de la forma de holgura no requiere un esfuerzo explícito, ya que los valores de A, b y c son los mismos tanto en la forma de holgura como en la estándar). Si la línea 2 encuentra que esta solución básica es factible, es decir, xNi 0 para todo i 2 N [ B—entonces la línea 3 devuelve la forma de holgura. De lo contrario, en la línea 4 formamos el programa lineal auxiliar Laux como en el Lema 29.11. Dado que la solución básica inicial de L no es factible, la solución básica inicial de la forma de holgura para Laux tampoco puede serlo. Para encontrar una solución factible básica, realizamos una sola operación de pivote. La línea 6 selecciona I D n C k como el índice de la variable básica que será la variable saliente en la próxima operación pivote.

Dado que las variables básicas son xnC1; xnC2; : : : ; xnCm, la variable saliente xl será la que tenga el valor más negativo. La línea 8 realiza esa llamada de PIVOT, con x0 entrando y xl saliendo. Veremos en breve que la solución básica resultante de esta llamada de PIVOT será factible. Ahora que tenemos una forma floja para la cual la solución básica es factible, podemos, en la línea 10, llamar repetidamente a PIVOT para resolver completamente el programa lineal auxiliar. Como demuestra la prueba en la línea 11, si encontramos una solución óptima para Laux con valor objetivo 0, entonces en las líneas 12 a 14, creamos una forma de holgura para L para la cual la solución básica es factible. Para hacerlo, primero, en las líneas 12 y 13, manejamos el caso degenerado en el que x0 aún puede ser básico con valor xN0 D 0. En este caso, realizamos un paso pivote para eliminar x0 de la base, usando cualquier e 2 N tal que a0e ≠ 0 como variable de entrada. La nueva solución básica sigue siendo factible; el pivote degenerado no cambia el valor de ninguna variable. A continuación, eliminamos todos los términos x0 de las restricciones y restauramos la función objetivo original para L. La función objetivo original puede contener variables básicas y no básicas. Por lo tanto, en la función objetivo reemplazamos cada variable básica por el lado derecho de su restricción asociada. La línea 15 luego devuelve esta forma de holgura modificada. Si, por otro lado, la línea 11 descubre que el programa lineal original L no es factible, entonces la línea 16 devuelve esta información.

Ahora demostramos la operación de INITIALIZE-SIMPLEX en el programa lineal (29.102)–(29.105). Este programa lineal es factible si podemos encontrar valores no negativos para x1 y x2 que satisfagan las desigualdades (29.103) y (29.104). Usando el Lema 29.11, formulamos el programa lineal auxiliar

$$\text{maximizar} \quad x_0 \quad (29.109)$$

$$\begin{array}{lllll} \text{sujeto a} & & & & \\ 2x_1 & x_2 & x_0 & 2 & (29.110) \\ x_1 & 5x_2 & x_0 & 4 & (29.111) \end{array}$$

$$x_1; x_2; x_0 \quad 0$$

Por el Lema 29.11, si el valor objetivo óptimo de este programa lineal auxiliar es 0, entonces el programa lineal original tiene una solución factible. Si el objetivo óptimo

valor de este programa lineal auxiliar es negativo, entonces el programa lineal original no tiene una solución factible.

Escribimos este programa lineal en forma holgada, obteniendo

$$\begin{array}{lll} D & & x_0 \\ x_3 D 2 x_4 D 4 & & 2x_1 C x_2 C x_0 \\ & & x_1 C 5x_2 C x_0 : \end{array}$$

Todavía no estamos fuera de peligro porque la solución básica, que establecería  $x_4 = 4$ , no es factible para este programa lineal auxiliar. Sin embargo, podemos, con una llamada a PIVOT, convertir esta forma floja en una en la que la solución básica sea factible. Como indica la línea 8, elegimos  $x_0$  como la variable de entrada. En la línea 6, elegimos como variable saliente  $x_4$ , que es la variable básica cuyo valor en la solución básica es más negativo. Despues de pivotar, tenemos la forma de holgura

$$\begin{array}{lll} ' D 4 & x_1 C 5x_2 & x_4 \\ x_0 D 4 C x_1 x_3 D 6 & 5x_2 C x_4 4x_2 C \\ & x_1 & x_4 : \end{array}$$

La solución básica asociada es  $.x_0; x_1; x_2; x_3; x_4/ = .4; 0; 0; 6; 0/$ , lo cual es factible. Ahora llamamos repetidamente a PIVOT hasta que obtengamos una solución óptima para  $L_{aux}$ . En este caso, una llamada a PIVOT con  $x_2$  entrando y  $x_0$  saliendo produce

$$\begin{array}{lll} D & & x_0 \\ x_2 D - & \frac{4}{5} & \underline{x_0} \\ x_3 D - & \frac{5}{5} & C \quad \frac{1}{5} \quad C \quad \frac{x_4}{5} \\ & & \frac{5}{4x_0} \quad \frac{1}{5} \quad C \quad \frac{x_4}{5} \end{array}$$

Esta forma holgada es la solución final al problema auxiliar. Como esta solución tiene  $x_0 = 0$ , sabemos que nuestro problema inicial era factible. Además, dado que  $x_0 = 0$ , podemos eliminarlo del conjunto de restricciones. Luego restauramos la función objetivo original, con las sustituciones apropiadas hechas para incluir solo variables no básicas. En nuestro ejemplo, obtenemos la función objetivo

$$2x_1 x_2 D 2x_1 - \frac{4}{5} \quad \frac{x_0}{5} C \quad \frac{x_1}{5} C \quad \frac{x_4}{5}$$

Haciendo  $x_0 = 0$  y simplificando, obtenemos la función objetivo

$$\frac{4}{5} \frac{9x_1}{5} - \frac{x_4}{5} :$$

y la forma floja

$$\begin{array}{rccccc}
 & 4 & & 9x_1 & & x_4 \\
 D & \underline{-} & C & \underline{\frac{9}{5}} & & \underline{\frac{x_4}{5}} \\
 & 5 & & 5 & & 5 \\
 & 4 & & x_1 & & x_4 \\
 x_2 D & - & C & \underline{\frac{x_1}{5}} & C & \underline{\frac{x_4}{5}} \\
 & 5 & 14 & 9x_1 & & x_4 \\
 x_3 D & \underline{-} & & \underline{\frac{5}{5}} & C & \underline{\frac{x_4}{5}}
 \end{array}$$

Esta forma de holgura tiene una solución básica factible y podemos devolverla al procedimiento SIMPLEX.

Ahora mostramos formalmente la corrección de INITIALIZE-SIMPLEX.

Lema 29.12 Si

un programa lineal L no tiene una solución factible, INITIALIZE-SIMPLEX devuelve "no factible". De lo contrario, devuelve una forma de holgura válida para la cual la solución básica es factible.

Prueba Primero suponga que el programa lineal L no tiene solución factible. Entonces, por el Lema 29.11, el valor objetivo óptimo de  $L_{\text{aux}}$ , definido en (29.106)–(29.108), es distinto de cero, y por la restricción de no negatividad de  $x_0$ , el valor objetivo óptimo debe ser negativo. Además, este valor objetivo debe ser finito, ya que haciendo  $x_i = 0$ , para  $i = 1, 2, \dots, n$ , y  $x_0 = \min f_{\text{bi}}$  es factible, y esta solución tiene valor objetivo  $\min f_{\text{bi}}$ . Por lo tanto, la línea 10 de INITIALIZE-SIMPLEX encuentra una solución con un valor objetivo no positivo. Sea  $x_N$  la solución básica asociada con la forma de holgura final. No podemos tener  $x_N = 0$ , porque entonces  $L_{\text{aux}}$  tendría valor objetivo 0, lo que contradice que el valor objetivo es negativo.

Por lo tanto, la prueba en la línea 11 da como resultado que la línea 16 devuelva "no factible".

Suponga ahora que el programa lineal L tiene una solución factible. Del ejercicio 29.3-4 sabemos que si  $b_i = 0$  para  $i = 1, 2, \dots, m$ , entonces la solución básica asociada con la forma de holgura inicial es factible. En este caso, las líneas 2 y 3 devuelven la forma de holgura asociada con la entrada. (Convertir la forma estándar a la forma floja es fácil, ya que A, b y c son iguales en ambos).

En el resto de la prueba, manejamos el caso en el que el programa lineal es factible pero no regresamos en la línea 3. Argumentamos que en este caso, las líneas 4 a 10 encuentran una solución factible para  $L_{\text{aux}}$  con valor objetivo 0. Primero, por las líneas 1-2, debemos tener

$b_k < 0$ ;

y

$b_i \leq 0$  para cada  $i = 1, 2, \dots, B$  (29.112)

En la línea 8, realizamos una operación de pivote en la que la variable de salida  $x_1$  (recuerde que  $i = 1, 2, \dots, B$ , de modo que  $b_i < 0$ ) es el lado izquierdo de la ecuación con mínimo  $b_i$ , y la variable de entrada es  $x_0$ , la variable extra añadida. ahora mostramos

que después de este pivote, todas las entradas de  $b$  son no negativas y, por lo tanto, la solución básica de  $L_{aux}$  es factible. Sea  $x_N$  la solución básica después de la llamada a PIVOT, y sea  $b_i$  y  $e$ . Por ser valores devueltos por PIVOT, el Lema 29.1 implica que

$$x_{Nj} \in D \quad (\text{bl} \geq 0 \text{ y } b_i \geq 0 \text{ para } i \neq j) \quad (29.113)$$

La llamada a PIVOT en la línea 8 tiene  $e \in D \setminus 0$ . Si reescribimos las desigualdades (29.107), para incluir los coeficientes  $a_{i0}$ ,

$$\sum_{j=0}^n a_{ij} x_j \leq b_i \text{ para } i = 1, 2, \dots, m; \quad (29.114)$$

entonces

$$a_{i0} \leq b_i \leq b_i + \sum_{j=1}^n a_{ij} x_j \text{ para cada } i = 1, 2, \dots, m. \quad (29.115)$$

(Observe que  $a_{i0}$  es el coeficiente de  $x_0$  tal como aparece en las desigualdades (29.114), no la negación del coeficiente, porque  $L_{aux}$  está en forma estándar y no holgada).

Como  $b_i \geq 0$ , también tenemos que  $a_{i0} \leq b_i \leq b_i + \sum_{j=1}^n a_{ij} x_j \leq b_i + \sum_{j=1}^n a_{ij} \cdot 0 = b_i$ . Por lo tanto,  $a_{i0} \leq b_i \leq b_i$ . Para las variables básicas restantes, tenemos

$$\begin{aligned} x_{Nj} &= a_{i0} \leq b_i \leq b_i + \sum_{j=1}^n a_{ij} x_j \quad (\text{por la ecuación (29.113)}) \\ &\leq b_i + \sum_{j=1}^n a_{ij} \cdot 0 = b_i \quad (\text{por la ecuación (29.115)}) \\ &\leq b_i \quad (\text{por la ecuación (29.115) y } a_{i0} \leq b_i) \\ &= b_i \quad (\text{por la desigualdad (29.112)}), \end{aligned}$$

lo que implica que cada variable básica ahora es no negativa. Por lo tanto, la solución básica después de la llamada a PIVOT en la línea 8 es factible. A continuación ejecutamos la línea 10, que resuelve  $L_{aux}$ . Como hemos supuesto que  $L$  tiene una solución factible, el Lema 29.11 implica que  $L_{aux}$  tiene una solución óptima con valor objetivo 0. Como todas las formas de holgura son equivalentes, la solución básica final de  $L_{aux}$  debe tener  $x_{N0} = 0$ , y después de quitar  $x_0$  de el programa lineal, obtenemos una forma holgada que es factible para  $L$ . La línea 15 luego devuelve esta forma de holgura. ■

### Teorema fundamental de la programación lineal

Concluimos este capítulo mostrando que el procedimiento SIMPLEX funciona. En particular, cualquier programa lineal no es factible, no está acotado o tiene una solución óptima con un valor objetivo finito. En cada caso, SIMPLEX actúa adecuadamente.

**Teorema 29.13 (Teorema fundamental de la programación lineal)**

Cualquier programa lineal  $L$ , dado en forma estándar, ya sea

1. tiene una solución óptima con un valor objetivo finito,
2. es inviable, o
3. es ilimitado.

Si  $L$  es inviable, SIMPLEX devuelve "no factible". Si  $L$  no tiene límites, SIMPLEX devuelve "sin límites". De lo contrario, SIMPLEX devuelve una solución óptima con un valor objetivo finito.

**Demostración** Por el Lema 29.12, si el programa lineal  $L$  no es factible, entonces SIMPLEX devuelve "no factible". Ahora suponga que el programa lineal  $L$  es factible. Por el Lema 29.12, INITIALIZE-SIMPLEX devuelve una forma floja para la cual la solución básica es factible.

Por el Lema 29.7, por lo tanto, SIMPLEX devuelve "ilimitado" o termina con una solución factible. Si termina con una solución finita, entonces el teorema 29.10 nos dice que esta solución es óptima. Por otro lado, si SIMPLEX devuelve "no acotado", el Lema 29.2 nos dice que el programa lineal  $L$  es realmente no acotado. Dado que SIMPLEX siempre termina en una de estas formas, la prueba está completa. ■

### Ejercicios

#### 29.5-1

Dé un pseudocódigo detallado para implementar las líneas 5 y 14 de INITIALIZE-SIMPLEX.

#### 29.5-2

Muestre que cuando INITIALIZE-SIMPLEX ejecuta el ciclo principal de SIMPLEX , nunca puede regresar "ilimitado".

#### 29.5-3

Suponga que tenemos un programa lineal  $L$  en forma estándar, y suponga que tanto para  $L$  como para el dual de  $L$ , las soluciones básicas asociadas con las formas iniciales de holgura son factibles. Demuestre que el valor objetivo óptimo de  $L$  es 0.

#### 29.5-4

Suponga que permitimos desigualdades estrictas en un programa lineal. Demuestre que en este caso, el teorema fundamental de la programación lineal no se cumple.

## 29.5-5

Resuelva el siguiente programa lineal usando SIMPLEX:

maximice  $x_1 + 3x_2$  sujeto a

$$\begin{array}{lll}
 x_1 & x_2 & 8 \\
 x_1 & x_2 & 3 \\
 x_1 + 4x_2 & & 2 \\
 x_1, x_2 & & 0
 \end{array}$$

## 29.5-6

Resuelva el siguiente programa lineal usando SIMPLEX:

maximizar  $x_1 + 2x_2$

sujeto a

$$\begin{array}{lll}
 x_1 + 2x_2 + 6x_3 & & 4 \\
 2x_1 & & 12 \\
 x_2 & & 1 \\
 x_1, x_2 & & 0
 \end{array}$$

## 29.5-7

Resuelva el siguiente programa lineal usando SIMPLEX:

maximice  $x_1 + 3x_2$  sujeto a

$$\begin{array}{lll}
 x_1 + x_2 & & 1 \\
 x_1 & x_2 & 3 \\
 x_1 + 4x_2 & & 2 \\
 x_1, x_2 & & 0
 \end{array}$$

## 29.5-8

Resuelva el programa lineal dado en (29.6)–(29.10).

## 29.5-9

Considere el siguiente programa lineal de 1 variable, al que llamamos P:

maximizar  $tx$

sujeto a

$$\begin{array}{ll}
 \text{receta} & s \\
 x & 0
 \end{array}$$

donde  $r, s$  y  $t$  son números reales arbitrarios. Sea D el dual de P.

Indique para qué valores de r, s y t puede afirmar que

1. Tanto P como D tienen soluciones óptimas con valores objetivos finitos.
2. P es factible, pero D no es factible.
3. D es factible, pero P no es factible.
4. Ni P ni D son factibles.

## Problemas

### 29-1 Viabilidad de la desigualdad lineal

Dado un conjunto de m desigualdades lineales en n variables  $x_1; x_2; \dots; x_n$ , el problema de viabilidad de la desigualdad lineal pregunta si existe un ajuste de las variables que satisfaga simultáneamente cada una de las desigualdades.

- a. Demuestre que si tenemos un algoritmo para programación lineal, podemos usarlo para resolver un problema de factibilidad de desigualdad lineal. El número de variables y restricciones que utilice en el problema de programación lineal debe ser polinomial en n y m.
- b. Demuestre que si tenemos un algoritmo para el problema de viabilidad de la desigualdad lineal, podemos usarlo para resolver un problema de programación lineal. El número de variables y desigualdades lineales que utilice en el problema de viabilidad de desigualdad lineal debe ser polinomial en n y m, el número de variables y restricciones en el problema lineal.

### 29-2 Holgura complementaria La

Holgura complementaria describe una relación entre los valores de las variables primarias y las restricciones duales y entre los valores de las variables duales y las restricciones primarias. Sea  $x_N$  una solución factible del programa lineal primal dado en (29.16)–(29.18), y sea  $y_N$  una solución factible del programa lineal dual dado en (29.83)–(29.85). La holgura complementaria establece que las siguientes condiciones son necesarias y suficientes para que  $x_N$  y  $y_N$  sean óptimos:

$$\sum_{j=1}^m a_{ij} y_{Ni} \leq b_j \quad \text{para } j = 1, 2, \dots, m$$

y

$$\sum_{i=1}^n a_{ij} x_{Nj} \geq c_i \quad \text{para } i = 1, 2, \dots, n$$

- a. Verifique que la holgura complementaria se cumple para el programa lineal en las líneas (29.53)–(29.57).
- b. Demuestre que la holgura complementaria se cumple para cualquier programa lineal primario y su dual correspondiente.
- C. Demuestre que una solución factible  $x_N$  de un programa lineal primal dado en las líneas (29.16)–(29.18) es óptima si y solo si existen valores  $y_N \in D$ ,  $y_{N1}; y_{N2}; \dots; y_{Nm}$  tal que
1.  $y_N$  es una solución factible del programa lineal dual dado en (29.83)–(29.85), 2.  $\sum_i a_{ij} y_{Ni} \leq c_j$  para todo  $j$  tal que  $x_{Nj} > 0$ , y 3.  $y_{Ni} \geq 0$  para todo  $i$  tal que  $\sum_j a_{ij} x_{Nj} < b_i$ .

### 29-3 Programación lineal entera Un

problema de programación lineal entera es un problema de programación lineal con la restricción adicional de que las variables  $x$  deben tomar valores enteros. El ejercicio 34.5-3 muestra que solo determinar si un programa lineal entero tiene una solución factible es NP-difícil, lo que significa que no existe un algoritmo de tiempo polinomial conocido para este problema.

- a. Demuestre que la dualidad débil (Lema 29.8) se cumple para un programa lineal entero.
- b. Muestre que la dualidad (Teorema 29.10) no siempre se cumple para un entero lineal programa.
- C. Dado un programa lineal primario en forma estándar, definamos  $P$  como el valor objetivo óptimo para el programa lineal primario,  $D$  como el valor objetivo óptimo para su dual,  $IP$  como el valor objetivo óptimo para la versión entera del programa primal (es decir, el primal con la restricción añadida de que las variables toman valores enteros), e  $ID$  como el valor objetivo óptimo para la versión entera del dual. Suponiendo que tanto el programa entero primal como el programa entero dual son factibles y acotados, demuestre que

$ID \leq IP \leq P \leq D \leq ID$

### 29-4 Lema de Farkas Sean

A una matriz  $m \times n$  y un vector  $n$ . Entonces el lema de Farkas establece que exactamente uno de los sistemas

Hacha 0  
 $cTx > 0$   
 $y$   
 $ATy \leq c$   
 $y \geq 0$

es solucionable, donde  $x$  es un  $n$ -vector y  $y$  es un  $m$ -vector. Demostrar el lema de Farkas.

### 29-5 Circulación de costo mínimo En

en este problema, consideraremos una variante del problema de flujo de costo mínimo de la sección 29.2 en el que no tenemos una demanda, una fuente o un sumidero. En cambio, se nos da, como antes, una red de flujo y costos de borde  $a_{uv}$ . Un flujo es factible si satisface la restricción de capacidad en cada borde y la conservación del flujo en cada vértice. El objetivo es encontrar, entre todos los flujos factibles, el de mínimo costo. Llamamos a este problema el problema de circulación de costo mínimo.

- a. Formule el problema de circulación de costo mínimo como un programa lineal.
- b. Supongamos que para todas las aristas  $u, v \in E$ , tenemos  $a_{uv} \geq 0$ . Caracterizar una solución óptima al problema de circulación de costo mínimo.
- C. Formule el problema de flujo máximo como un programa lineal de problema de circulación de costo mínimo. A eso se le da una instancia de problema de flujo máximo  $G = (V, E)$  con fuente  $s$ , sumidero  $t$  y capacidades de borde  $c$ , cree un problema de circulación de costo mínimo dando una red (posiblemente diferente)  $G' = (V, E')$  con capacidades de borde  $c'$  y costos de borde  $a'$  tal que pueda discernir una solución al problema de flujo máximo de una solución al problema de circulación de costo mínimo.
- d. Formule el problema del camino más corto de fuente única como un programa lineal de problema de circulación de costo mínimo.

#### Notas del capítulo

Este capítulo solo comienza a estudiar el amplio campo de la programación lineal. Varios libros están dedicados exclusivamente a la programación lineal, incluidos los de Chvátal [69], Gass [130], Karloff [197], Schrijver [303] y Vanderbei [344]. Muchos otros libros brindan una buena cobertura de la programación lineal, incluidos los de Papadimitriou y Steiglitz [271] y Ahuja, Magnanti y Orlin [7]. La cobertura de este capítulo se basa en el enfoque adoptado por Chvátal.

El algoritmo simplex para programación lineal fue inventado por G. Dantzig en 1947. Poco después, los investigadores descubrieron cómo formular una serie de problemas en una variedad de campos como programas lineales y resolverlos con el algoritmo simplex. Como resultado, florecieron las aplicaciones de la programación lineal, junto con varios algoritmos. Las variantes del algoritmo simplex siguen siendo los métodos más populares para resolver problemas de programación lineal. Esta historia aparece en varios lugares, incluidas las notas en [69] y [197].

El algoritmo elíptico fue el primer algoritmo de tiempo polinomial para programación lineal y se debe a LG Khachian en 1979; se basó en trabajos anteriores de NZ Shor, DB Judin y AS Nemirovskii. Grötschel, Lovász y Schrijver [154] describen cómo usar el algoritmo elíptico para resolver una variedad de problemas en la optimización combinatoria. Hasta la fecha, el algoritmo elíptico no parece ser competitivo con el algoritmo simplex en la práctica.

El artículo de Karmarkar [198] incluye una descripción del primer algoritmo de punto interior. Muchos investigadores posteriores diseñaron algoritmos de punto interior.

Buenas encuestas aparecen en el artículo de Goldfarb y Todd [141] y en el libro de Ye [361].

El análisis del algoritmo simplex sigue siendo un área activa de investigación. V. Klee y GJ Minty construyeron un ejemplo en el que el algoritmo simplex se ejecuta a través de  $2n - 1$  iteraciones. El algoritmo simplex suele funcionar muy bien en la práctica y muchos investigadores han tratado de dar una justificación teórica para esta observación empírica. Una línea de investigación iniciada por KH Borgwardt, y continuada por muchos otros, muestra que bajo ciertas suposiciones probabilísticas sobre la entrada, el algoritmo simplex converge en el tiempo polinomial esperado. Spielman y Teng [322] avanzaron en esta área, introduciendo el “análisis suavizado de algoritmos” y aplicándolo al algoritmo simplex.

Se sabe que el algoritmo simplex se ejecuta de manera eficiente en ciertos casos especiales. Particularmente digno de mención es el algoritmo de red simplex, que es el algoritmo simplex, especializado en problemas de flujo de red. Para ciertos problemas de red, incluidos los problemas de rutas más cortas, flujo máximo y flujo de costo mínimo, las variantes del algoritmo de red simple se ejecutan en tiempo polinomial. Véase, por ejemplo, el artículo de Orlin [268] y las citas del mismo.

---

## 30

## Polinomios y la FFT

El método directo de sumar dos polinomios de grado  $n$  toma  $.n^2$  tiempo, pero el método directo de multiplicarlos toma  $.n^3$  tiempo. En este capítulo, mostraremos cómo la transformada rápida de Fourier, o FFT, puede reducir el tiempo para multiplicar polinomios a  $.n \lg n$ .

El uso más común de las transformadas de Fourier y, por lo tanto, de la FFT, es en el procesamiento de señales. Una señal se da en el dominio del tiempo: como una función que asigna el tiempo a la amplitud. El análisis de Fourier nos permite expresar la señal como una suma ponderada de sinusoides desfasadas de frecuencias variables. Los pesos y fases asociados con las frecuencias caracterizan la señal en el dominio de la frecuencia. Entre las muchas aplicaciones cotidianas de FFT se encuentran las técnicas de compresión utilizadas para codificar información de audio y video digital, incluidos los archivos MP3. Varios libros excelentes profundizan en la rica área del procesamiento de señales; las notas del capítulo hacen referencia a algunos

polinomios

Un polinomio en la variable  $x$  sobre un cuerpo algebraico  $F$  representa una función  $Ax/$  como una suma formal:

$$\sum_{j=0}^{n-1} a_j x^j$$

Llamamos a los valores  $a_0, a_1, \dots, a_n$  los coeficientes del polinomio. Los coeficientes se extraen de un campo  $F$ , típicamente el conjunto  $C$  de números complejos. Un polinomio  $Ax/$  tiene grado  $k$  si su coeficiente distinto de cero más alto es  $a_k$ ; escribimos ese grado  $A/ D k$ . Cualquier número entero estrictamente mayor que el grado de un polinomio es un límite de grado de ese polinomio. Por lo tanto, el grado de un polinomio de grado  $n$  puede ser cualquier número entero entre 0 y  $n$ , inclusive.

Podemos definir una variedad de operaciones en polinomios. Para la suma de polinomios, si  $Ax/$  y  $Bx/$  son polinomios de grado  $n$ , su suma es un polinomio.

mial  $Cx^j$ , también de  $n$  ligada a grados, tal que  $Cx^j D Ax^k C Bx^l$  para todo  $x$  en el campo subyacente. es decir, si

$$Hacha/ D Xn1 a_j x^j$$

$j \leq 0$

y

$$Bx/D Xn1 b_j x^j ;$$

$j \leq 0$

entonces

$$Cx/D Xn1 c_j x^j ;$$

$j \leq 0$

donde  $c_j = a_j b_j$  para  $j = 0, 1, \dots, n$ . Por ejemplo, si tenemos los polinomios  $Ax^j / D 6x^3 C 7x^2 10x C 9$  y  $Bx^j / D 2x^3 C 4x^5$ , entonces  $Cx^j / D 4x^3 C 7x^2 6x C 4$ .

Para la multiplicación de polinomios, si  $Ax^j$  y  $Bx^j$  son polinomios de grado  $n$ , su producto  $Cx^j$  es un polinomio de grado  $2n + 1$  tal que  $Cx^j / D Ax^k Bx^l$  para todo  $x$  en el campo subyacente. Probablemente hayas multiplicado polinomios anteriormente, multiplicando cada término en  $Ax^j$  por cada término en  $Bx^j$  y luego combinando términos con potencias iguales. Por ejemplo, podemos multiplicar  $Ax^j / D 6x^3 C 7x^2 10x C 9$  y  $Bx^j / D 2x^3 C 4x^5$  de la siguiente manera:

$$\begin{array}{r}
 6x^3 C 7x^2 10x C 9 \\
 2x^3 \quad \quad \quad C 4x^5 \\
 \hline
 30x^3 35x^2 C 50x^4 \\
 24x^4 C 28x^3 40x^2 C \\
 \hline
 36x^12x^6 14x^5 C 20x^4 \\
 \hline
 18x^3 12x^6 14x^5 C 44x^4 20x^3 75x^2 C 86x^45
 \end{array}$$

Otra forma de expresar el producto  $Cx^j$  es

$$Cx/D \sum_{j=0}^{2n+1} c_j x^j ; \tag{30.1}$$

dónde

$$c_j = \sum_{k=0}^n a_k b_{j-k} \tag{30.2}$$

Tenga en cuenta que grado.C / D grado.A/ C grado.B/, lo que implica que si A es un polinomio de grados enlazados na y B es un polinomio de grados enlazados nb, entonces C es un polinomio de grados enlazados na C nb 1. Dado que un polinomio de grado k es también un polinomio de grado k C 1, normalmente diremos que el polinomio producto C es un polinomio de grado na C nb.

### Bosquejo del capítulo

La sección 30.1 presenta dos formas de representar polinomios: la representación de coeficiente y la representación de valor de punto. Los métodos sencillos para multiplicar polinomios, las ecuaciones (30.1) y (30.2), toman  $\sim n^2$  tiempo cuando representamos polinomios en forma de coeficiente, pero solo  $\sim n$  tiempo cuando los representamos en forma de valor puntual. Sin embargo, podemos multiplicar polinomios usando la representación del coeficiente en solo  $\sim n \lg n$  tiempo al convertir entre las dos representaciones. Para ver por qué funciona este enfoque, primero debemos estudiar las raíces complejas de la unidad, lo que hacemos en la Sección 30.2. Luego, usamos la FFT y su inversa, también descritas en la Sección 30.2, para realizar las conversiones. La Sección 30.3 muestra cómo implementar la FFT rápidamente en modelos tanto en serie como en paralelo.

Este capítulo utiliza extensamente números complejos, y dentro de este capítulo usamos el símbolo  $i$  exclusivamente para denotar  $p_1$ .

## 30.1 Representación de polinomios

Las representaciones de coeficientes y valores puntuales de los polinomios son, en cierto sentido, equivalentes; es decir, un polinomio en forma de valor puntual tiene una contrapartida única en forma de coeficiente. En esta sección, presentamos las dos representaciones y mostramos cómo combinarlas para que podamos multiplicar dos polinomios  $n$  con límites de grado en  $\sim n \lg n$  tiempo.

### Representación del coeficiente

Una representación de coeficientes de un polinomio  $Ax/ D Pn1 jD0 aj xj$  de límite grado  $n$  es un vector de coeficientes a D  $.a_0; a_1; \dots; a_n/$ . En las ecuaciones matriciales de este capítulo, generalmente trataremos los vectores como vectores columna.

La representación de coeficientes es conveniente para ciertas operaciones sobre polinomios. Por ejemplo, la operación de evaluar el polinomio  $Ax/$  en un punto dado  $x_0$  consiste en calcular el valor de  $A.x_0/$ . Podemos evaluar un polinomio en tiempo  $\sim n$  usando la regla de Horner:

$$A.x_0/ \equiv a_0 + a_1 x_0 + a_2 x_0^2 + \dots + a_n x_0^n$$

De manera similar, sumando dos polinomios representados por los vectores de coeficientes a D .a0; a1;:::;an1/ y b D .b0; b1;:::;bn1/ toma „n/ tiempo: simplemente producimos el vector coeficiente c D .c0; c1;:::;cn1/, donde  $c_j = a_j b_j$  para  $j = 0, 1, \dots, n-1$

Ahora, considere multiplicar dos polinomios Ax/ y Bx/ ligados a grados representados en forma de coeficiente. Si usamos el método descrito por las ecuaciones (30.1) y (30.2), la multiplicación de polinomios toma un tiempo „n<sup>2</sup>/, ya que debemos multiplicar cada coeficiente del vector a por cada coeficiente del vector b. La operación de multiplicar polinomios en forma de coeficiente parece ser considerablemente más difícil que evaluar un polinomio o sumar dos polinomios. El vector de coeficientes c resultante, dado por la ecuación (30.2), también se denomina convolución de los vectores de entrada a y b, denotados c D a “ b. Dado que la multiplicación de polinomios y el cálculo de convoluciones son problemas computacionales fundamentales de considerable importancia práctica, este capítulo se concentra en algoritmos eficientes para ellos.

### Representación de valor de punto

Una representación de valor puntual de un polinomio Ax/ de n ligado a grados es un conjunto de n pares de valores puntuales

$f(x_0); y_0; f(x_1); y_1; \dots; f(x_n); y_n$

tal que todos los  $x_k$  son distintos y

$$y_k = f(x_k) \quad (30.3)$$

para  $k = 0, 1, \dots, n-1$ . Un polinomio tiene muchas representaciones diferentes de valores puntuales, ya que podemos usar cualquier conjunto de n puntos distintos  $x_0; x_1; \dots; x_n$  como base para la representación.

Calcular una representación de valor puntual para un polinomio dado en forma de coeficiente es, en principio, sencillo, ya que todo lo que tenemos que hacer es seleccionar n puntos distintos  $x_0; x_1; \dots; x_n$  y luego evalúe  $A(x_k)$  para  $k = 0, 1, \dots, n-1$ . El método de Horner, evaluar un polinomio en n puntos toma un tiempo „n<sup>2</sup>/ . Veremos más adelante que si elegimos los puntos  $x_k$  hábilmente, podemos acelerar este cálculo para que se ejecute en el tiempo „n lg n/.

La inversa de la evaluación, que determina la forma del coeficiente de un polinomio a partir de una representación de valor puntual, es la interpolación. El siguiente teorema muestra que la interpolación está bien definida cuando el polinomio de interpolación deseado debe tener un límite de grado igual al número dado de pares de puntos y valores.

#### Teorema 30.1 (Unicidad de un polinomio interpolador)

Para cualquier conjunto  $f(x_0); y_0; f(x_1); y_1; \dots; f(x_n); y_n$  de n pares de valores puntuales tales que todos los valores de  $x_k$  son distintos, hay un polinomio único  $A(x)$  de n ligado a grados tal que  $y_k = A(x_k)$  para  $k = 0, 1, \dots, n-1$

**Prueba** La prueba se basa en la existencia de la inversa de una determinada matriz. La ecuación (30.3) es equivalente a la ecuación matricial

$$\begin{matrix} & x_0 & x_1 & \dots & x_n \\ \begin{matrix} x_0 & x_1 & \dots & x_n \\ x_1 & x_2 & \dots & x_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_0 & x_1 & \dots & x_n \end{matrix} & \cdot \begin{matrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{matrix} & = & \begin{matrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{matrix} \end{matrix} \quad (30.4)$$

La matriz de la izquierda se denota  $V[x_0; x_1; \dots; x_n]$  y se conoce como Vander matriz mundial. Por el Problema D-1, esta matriz tiene determinante

$$\prod_{k=0}^{n-1} (x_k - x_j) \neq 0$$

y por tanto, por el teorema D.5, es invertible (es decir, no singular) si las  $x_k$  son distintas. Por lo tanto, podemos resolver los coeficientes  $a_j$  únicamente dada la representación de valor puntual: a

$$DV[x_0; x_1; \dots; x_n] = y$$

La demostración del Teorema 30.1 describe un algoritmo de interpolación basado en resolver el conjunto (30.4) de ecuaciones lineales. Usando los algoritmos de descomposición LU del Capítulo 28, podemos resolver estas ecuaciones en el tiempo  $O(n^3)$ .

Un algoritmo más rápido para la interpolación de  $n$  puntos se basa en la fórmula de Lagrange:

$$y = \frac{\sum_{j=0}^{n-1} y_j \prod_{k=0, k \neq j}^{n-1} (x - x_k)}{\prod_{k=0}^{n-1} (x_k - x_j)} \quad (30.5)$$

Es posible que desee verificar que el lado derecho de la ecuación (30.5) es un polinomio de grado  $n$  que satisface  $A(x_k) = y_k$  para todo  $k$ . El ejercicio 30.1-5 le pregunta cómo calcular los coeficientes  $A$  usando la fórmula de Lagrange en el tiempo  $O(n^2)$ .

Por lo tanto, la evaluación e interpolación de  $n$  puntos son operaciones inversas bien definidas que se transforman entre la representación del coeficiente de un polinomio y la representación del valor de un punto.<sup>1</sup> Los algoritmos descritos anteriormente para estos problemas toman el tiempo  $O(n^2)$ .

La representación de valor puntual es bastante conveniente para muchas operaciones con polinomios. Para la suma, si  $Cx = Ax + Bx$ , entonces  $C(x) = A(x) + B(x)$  para cualquier punto  $x$ . Más precisamente, si tenemos una representación de valor puntual para  $A$ ,

<sup>1</sup>La interpolación es un problema notoriamente complicado desde el punto de vista de la estabilidad numérica. Aunque los enfoques descritos aquí son matemáticamente correctos, las pequeñas diferencias en las entradas o los errores de redondeo durante el cálculo pueden causar grandes diferencias en el resultado.

$f.x_0; y_0/; x_1; y_1/; \dots; x_n; y_n/g ;$

y para B,

$f.x_0; y_0/; x_1; y_1/; \dots; x_n; y_n/g$

(nótese que A y B se evalúan en los mismos n puntos), entonces una representación de valor de punto para C es

$f.x_0; y_0 C y_0/; x_1; y_1 C y_1/; \dots; x_n; y_n C y_n/g :$

Por lo tanto, el tiempo para sumar dos polinomios de grado n en forma de valor puntual es  $,n/$ .

De manera similar, la representación de valor puntual es conveniente para multiplicar polinomios. Si  $Cx/D Ax/Bx/$ , entonces  $C.xk/D A.xk/B.xk/$  para cualquier punto  $xk$ , y podemos multiplicar puntualmente una representación de valor de punto para A por una representación de valor de punto para B para obtener una representación de valor en puntos para C. Sin embargo, debemos enfrentar el problema de que  $\text{grado.}C / D \text{ grado.}A / C \text{ grado.}B/$ ; si A y B son de grado n, entonces C es de grado  $2n$ . Una representación estándar de valores puntuales para A y B consta de n pares de valores puntuales para cada polinomio. Cuando los multiplicamos juntos, obtenemos  $n$  pares de valores puntuales, pero necesitamos  $2n$  pares para interpolar un polinomio único C de  $2n$  ligado a grados. (Consulte el ejercicio 30.1-4.) Por lo tanto, debemos comenzar con representaciones de valores puntuales "extendidas" para A y B que constan de  $2n$  pares de valores puntuales cada una. Dada una representación extendida de valores puntuales para A,

$f.x_0; y_0/; x_1; y_1/; \dots; x_{2n}; y_{2n}/g ;$

y una correspondiente representación extendida de valores puntuales

para B,  $f.x_0; y_0/; x_1; y_1/; \dots; x_{2n}; y_{2n}/g ;$

entonces una representación de valor puntual para C es

$f.x_0; y_0y_0/; x_1; y_1y_1/; \dots; x_{2n}; y_{2n}y_{2n}/g :$

Dados dos polinomios de entrada en forma de valor de punto extendida, vemos que el tiempo para multiplicarlos para obtener la forma de valor de punto del resultado es  $,n/$ , mucho menos que el tiempo requerido para multiplicar polinomios en forma de coeficiente.

Finalmente, consideraremos cómo evaluar un polinomio dado en forma de valor puntual en un nuevo punto. Para este problema, no conocemos un enfoque más simple que convertir primero el polinomio en forma de coeficiente y luego evaluarlo en el nuevo punto.

Multiplicación rápida de polinomios en forma de coeficiente

¿Podemos usar el método de multiplicación de tiempo lineal para polinomios en forma de valor puntual para acelerar la multiplicación de polinomios en forma de coeficiente? La respuesta depende

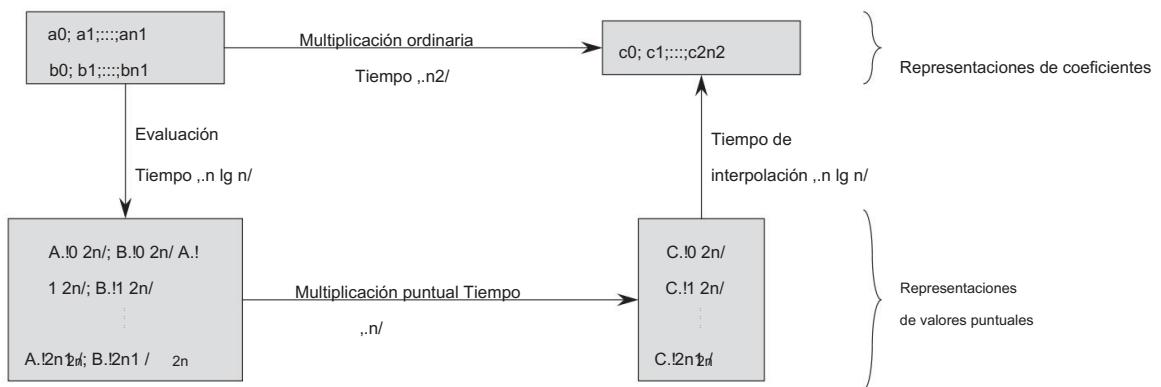


Figura 30.1 Esquema gráfico de un proceso eficiente de multiplicación de polinomios. Las representaciones en la parte superior están en forma de coeficiente, mientras que las de la parte inferior están en forma de valor de punto. Las flechas de izquierda a derecha corresponden a la operación de multiplicación. Los términos  $l2n$  son raíces complejas  $.2n^{th}$  de la unidad.

sobre si podemos convertir un polinomio rápidamente de forma de coeficiente a forma de valor puntual (evaluar) y viceversa (interpolación).

Podemos usar cualquier punto que queramos como puntos de evaluación, pero al elegir los puntos de evaluación con cuidado, podemos convertir entre representaciones en solo  $,n \lg n$  de tiempo. Como veremos en la Sección 30.2, si elegimos “raíces complejas de la unidad” como puntos de evaluación, podemos producir una representación de valor de punto tomando la transformada discreta de Fourier (o DFT) de un vector de coeficientes. Podemos realizar la operación inversa, la interpolación, tomando la “DFT inversa” de pares de puntos y valores, lo que produce un vector de coeficientes. La sección 30.2 mostrará cómo la FFT realiza las operaciones DFT y DFT inversa en  $,n \lg n$  tiempo.

La figura 30.1 muestra gráficamente esta estrategia. Un detalle menor se refiere a los límites de grado. El producto de dos polinomios de grado  $n$  es un polinomio de grado  $2n$ . Por lo tanto, antes de evaluar los polinomios de entrada  $A$  y  $B$ , primero duplicamos sus límites de grado a  $2n$  sumando  $n$  coeficientes de orden superior de 0.

Debido a que los vectores tienen  $2n$  elementos, usamos “raíces complejas  $.2n^{th}$  de la unidad”, que se denotan con los términos  $l2n$  en la figura 30.1.

Dada la FFT, tenemos el siguiente procedimiento  $,n \lg n$ -time para multiplicar dos polinomios  $Ax/$  y  $Bx/$  de grado  $n$  enlazado, donde las representaciones de entrada y salida están en forma de coeficiente. Suponemos que  $n$  es una potencia de 2; siempre podemos cumplir con este requisito agregando coeficientes cero de alto orden.

1. Límite de doble grado: cree representaciones de coeficientes de  $Ax/$  y  $Bx/$  como polinomios  $2n$  vinculados a grados agregando  $n$  coeficientes cero de alto orden a cada uno.

2. Evaluar: Calcule las representaciones de valores puntuales de  $Ax/$  y  $Bx/$  de longitud  $2n$  aplicando la FFT de orden  $2n$  en cada polinomio. Estas representaciones contienen los valores de los dos polinomios en las raíces  $.2n/th$  de la unidad.
3. Multiplicación puntual: calcule una representación de valor puntual para el polinomio  $Cx/$   $D Ax/$   $Bx/$  multiplicando estos valores juntos por puntos. Esta representación contiene el valor de  $Cx/$  en cada raíz  $.2n/ésima$  de la unidad.
4. Interpolar: Cree la representación del coeficiente del polinomio  $Cx/$  aplicando la FFT en  $2n$  pares de valores puntuales para calcular la DFT inversa.

Los pasos (1) y (3) toman tiempo  $,n/$ , y los pasos (2) y (4) toman tiempo  $,n \lg n/$ . Así, una vez que mostremos cómo usar la FFT, habremos probado lo siguiente.

#### Teorema 30.2

Podemos multiplicar dos polinomios de grado  $n$  en el tiempo  $,n \lg n/$ , con las representaciones de entrada y salida en forma de coeficiente. ■

### Ejercicios

#### 30.1-1

Multiplique los polinomios  $Ax/$   $D 7x^3 x^2 C x 10$  y  $Bx/$   $D 8x^3 6x C 3$  usando las ecuaciones (30.1) y (30.2).

#### 30.1-2

Otra forma de evaluar un polinomio  $Ax/$  de grado  $n$  en un punto dado  $x_0$  es dividir  $Ax/$  entre el polinomio  $.x x_0/$ , obteniendo un polinomio cociente  $qx/$  de grado  $n - 1$  y un resto  $r$ , tal que

$$Ax/ = qx/.x x_0/ + r$$

Claramente,  $A.x_0/ = DR r$ . Muestre cómo calcular el resto  $r$  y los coeficientes de  $qx/$  en el tiempo  $,n/$  a partir de  $x_0$  y los coeficientes de  $A$ .

#### 30.1-3

Derive una representación de valor de punto para  $Arev.x/$   $D \prod_{j=0}^{n-1} a_{n-j} x^j$  a partir de una representación de valor de punto para  $Ax/$   $D \prod_{j=0}^{n-1} a_j x^j$ , suponiendo que ninguno de los puntos es 0.

#### 30.1-4

Demuestre que se necesitan  $n$  pares de valores puntuales distintos para especificar de forma única un polinomio de  $n$  ligado a grados, es decir, si se dan menos de  $n$  pares de valores puntuales distintos, no logran especificar un polinomio único de grados- encuadrado (Sugerencia: usando el Teorema 30.1, ¿qué puede decir sobre un conjunto de  $n - 1$  pares de valores puntuales a los que agrega un par más de valores puntuales elegido arbitrariamente?)

## 30.1-5

Muestre cómo usar la ecuación (30.5) para interpolar en el tiempo  $,n^2/$ . (Sugerencia: primero calcule la representación del coeficiente del polinomio  $Q .x - x_j / y$  y luego dividir por  $j x_k$  según sea necesario para el numerador de cada término; consulte el ejercicio 30.1-2. Puede calcular cada uno de los  $n$  denominadores en el tiempo  $O(n)$ ).

## 30.1-6

Explique lo que está mal con el enfoque "obvio" de la división de polinomios usando una representación de valor de punto, es decir, dividiendo los valores de  $y$  y correspondientes. Discuta por separado el caso en el que la división sale exactamente y el caso en el que no sale.

## 30.1-7

Considere dos conjuntos  $A$  y  $B$ , cada uno con  $n$  enteros en el rango de 0 a  $10n$ . Deseamos calcular la suma cartesiana de  $A$  y  $B$ , definida por  $\sum_{x \in A} \sum_{y \in B} f(x, y)$

$A$  y  $y \in B$  : Tenga en cuenta que los

números enteros en  $C$  están en el rango de 0 a  $20n$ . Queremos encontrar los elementos de  $C$  y el número de veces que cada elemento de  $C$  se realiza como una suma de elementos en  $A$  y  $B$ . Muestre cómo resolver el problema en  $O(n \lg n)$  tiempo. (Sugerencia: represente  $A$  y  $B$  como polinomios de grado máximo  $10n$ ).

## 30.2 La DFT y la FFT

En la Sección 30.1, afirmamos que si usamos raíces unitarias complejas, podemos evaluar e interpolar polinomios en  $,n \lg n$  tiempo. En esta sección, definimos raíces complejas de la unidad y estudiamos sus propiedades, definimos la DFT y luego mostramos cómo la FFT calcula la DFT y su inversa en  $,n \lg n$  tiempo.

## Raíces complejas de unidad

Una raíz enésima compleja de la unidad es un número complejo! tal que

$e^{2\pi i k/n}$  para  $k = 0, 1, \dots, n-1$

Hay exactamente  $n$  raíces  $n$ -ésimas complejas de la unidad:  $e^{2\pi i k/n}$  para  $k = 0, 1, \dots, n-1$

Para interpretar esta fórmula, usamos la definición de la exponencial de un número complejo:

$\cos(u) + i \sin(u)$  : La

figura 30.2 muestra que las  $n$  raíces complejas de la unidad están igualmente espaciadas alrededor del círculo de radio unitario centrado en el origen del plano complejo. El valor

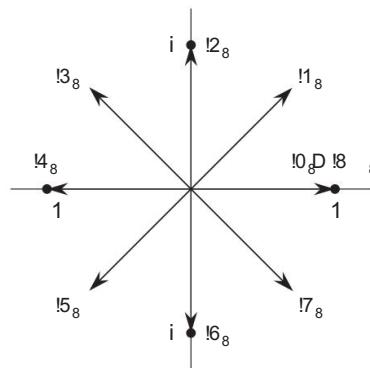


Figura 30.2 Los valores de  $\ln D$  cipal  $n=8$ ;  $!1_8, \dots, !7_8$  en el plano complejo, donde  $\sqrt[n]{D}$  es la raíz octava de la unidad.

$$\ln D \text{ e2 } i=n \quad (30.6)$$

es la raíz  $n$ -ésima principal de la unidad; de  $\ln$ . <sup>2</sup> todas las demás raíces enésimas complejas de la unidad son potencias

Las  $n$  raíces  $n$ -ésimas complejas de la unidad,  $\ln$

norte;  $!1_n, \dots, !n_1$ ;

formar un grupo bajo la multiplicación (ver Sección 31.3). Este grupo tiene la misma estructura que el grupo aditivo  $Z_n$ ; C/ módulo  $n$ , ya que  $\ln D = 1$  implica que  $\ln n!R \in D \cup C_k D \mod n$ . De manera similar,  $\ln$  proporciona algunas propiedades  $D \ln 1$ . los siguientes lemas esenciales de las raíces  $n$ -ésimas complejas de la unidad.

### Lema 30.3 (Lema de cancelación)

Para cualquier número entero  $n \neq 0$ ,  $k \neq 0$  y  $d > 0$ ,  $\ln k = \ln n$

$$\ln n = \ln d + \ln k \quad (30.7)$$

Prueba El lema se sigue directamente de la ecuación (30.6), ya que

$$\ln n = \ln d + \ln k$$

<sup>2</sup>Muchos otros autores definen  $\ln$  de manera diferente:  $\ln D = n$ . Esta definición alternativa tiende a usarse para aplicaciones de procesamiento de señales. Las matemáticas subyacentes son sustancialmente las mismas con cualquiera de las definiciones de  $\ln$ .

## Corolario 30.4

Para cualquier entero par  $n > 0$ ,

$$\|n=2\| \leq D \|2\| \leq 1$$

Prueba La prueba se deja como Ejercicio 30.2-1. ■

## Lema 30.5 (Lema reducido a la mitad)

Si  $n > 0$  es par, entonces los cuadrados de las  $n$  raíces enésimas complejas de la unidad son las raíces  $n/2$  complejas  $n/2$ -ésimas de la unidad.

Prueba Por el lema de cancelación, tenemos  $\|k/2\| \leq \|k\|$  para cualquier  $n=2$  no negativo, entero  $k$ . Tenga en cuenta que si elevamos al cuadrado todas las raíces  $n$ -ésimas complejas de la unidad, entonces obtenemos cada  $n/2$ -ésima raíz de la unidad exactamente dos veces, ya que

$$\begin{aligned} \|kCn=2\|^2 &\leq D \|2kCn\| \\ &\leq D \|2k\| \\ &\leq D \|k\|^2 \end{aligned}$$

Por tanto,  $\|k\|$  y  $\|kCn=2\|$  tienen el mismo cuadrado. También podríamos haber usado el Corolario 30.4 para probar esta propiedad, ya que  $\|n=2\| \leq D \|1\|$  implica  $\|kCn=2\| \leq D \|k\|$  por lo tanto  $\|kCn=2\|^2 \leq D^2 \|k\|^2$ . ■

Como veremos, el lema de reducción a la mitad es esencial para nuestro enfoque de divide y vencerás para convertir entre representaciones de polinomios de coeficientes y valores puntuales, ya que garantiza que los subproblemas recursivos son solo la mitad de grandes.

## Lema 30.6 (Lema de suma)

Para cualquier entero  $n \neq 1$  y entero  $k$  distinto de cero no divisible por  $n$ ,

$$\begin{aligned} X_{n1} &\leq \|k\| \leq D \|0\| \\ jD0 \end{aligned}$$

La ecuación de prueba (A.5) se aplica tanto a valores complejos como a valores reales, por lo que tenemos

$$\begin{aligned}
 Xn1 &= \frac{\sum_{j=0}^n a_j e^{-j\frac{2\pi}{n}k}}{n} \\
 &= \frac{\sum_{j=0}^n a_j e^{-j\frac{2\pi}{n}k}}{1} \\
 &= \frac{\sum_{j=0}^n a_j e^{-j\frac{2\pi}{n}k}}{1} \\
 &= \dots \\
 &\stackrel{D}{=} 0
 \end{aligned}$$

Debido a que requerimos que  $k$  no sea divisible por  $n$ , y debido a que  $\frac{2\pi k}{n}$  es divisible por  $n$ , nos aseguramos de que el denominador no sea 0. ■

### El DFT

Recuerde que deseamos evaluar un polinomio

$$\text{Hacha/ } D \sum_{j=0}^n a_j x^j$$

de  $n$  ligada a grados en  $10$  (es decir en las raíces  $n$ -ésimas complejas de la unidad).<sup>3</sup> Suponemos que  $A$  se da en forma de coeficiente:  $a = [a_0; a_1; \dots; a_n]$ . Definamos los resultados  $y_k$ , para  $k = 0; 1; \dots; n-1$ , por

$$y_k = \sum_{j=0}^n a_j e^{-j\frac{2\pi}{n}k}$$

$$Xn1 = \frac{\sum_{j=0}^n a_j e^{-j\frac{2\pi}{n}k}}{n} \quad (30.8)$$

El vector  $y = [y_0; y_1; \dots; y_{n-1}]$  es la transformada discreta de Fourier (DFT) del vector de coeficientes  $a = [a_0; a_1; \dots; a_n]$ . También escribimos  $y = DFTn.a$ .

### La FFT

Mediante el uso de un método conocido como transformada rápida de Fourier (FFT), que aprovecha las propiedades especiales de las raíces complejas de la unidad, podemos calcular  $DFTn.a$  en el tiempo  $\sim n \lg n$ , a diferencia del  $\sim n^2$  tiempo del método sencillo. Asumimos en todo momento que  $n$  es una potencia exacta de 2. Aunque las estrategias

<sup>3</sup>La longitud  $n$  es en realidad lo que llamamos  $2n$  en la Sección 30.1, ya que duplicamos el límite de grado de los polinomios dados antes de la evaluación. En el contexto de la multiplicación de polinomios, por lo tanto, en realidad estamos trabajando con raíces complejas  $\sqrt[2n]{th}$  de la unidad.

para tratar con tamaños que no son potencias de 2, están más allá del alcance de este libro.

El método FFT emplea una estrategia de divide y vencerás, usando los coeficientes indexados pares e impares de  $Ax/$  por separado para definir los dos nuevos polinomios  $A0.x/$  y  $A1.x/$  de grados enlazados  $n=2$ :

$$\begin{aligned} A0.x/ &= D a0 C a2x C a4x2 CC an2xn=21 ; \\ A1.x/ &= D a1 C \\ &a3x C a5x2 CC an1xn=21 : \end{aligned}$$

Tenga en cuenta que  $A0$  contiene todos los coeficientes indexados pares de  $A$  (la representación binaria del índice termina en 0) y  $A1$  contiene todos los coeficientes indexados impares (la representación binaria del índice termina en 1). Resulta que

$$Ax/ = A0.x/ + A1.x/ ; \quad (30.9)$$

de modo que el problema de evaluar  $Ax/$  en  $n=1$  se reduce a

- evaluar los polinomios  $A0.x/$  y  $A1.x/$  con límite de grados en los puntos

$$\dots, n/2^0, n/2^1, \dots, n/2^{\lfloor \log_2 n \rfloor}, \quad (30.10)$$

y luego

- combinar los resultados según la ecuación (30.9).

Por el lema de reducción a la mitad, la lista de valores (30.10) no consta de  $n$  valores distintos, sino sólo de  $n=2$  raíces complejas  $n=2/\text{ésimas}$  de la unidad, donde cada raíz aparece exactamente dos veces. Por lo tanto, evaluamos recursivamente los polinomios  $A0$  y  $A1$  de  $n=2$  ligados a grados en el complejo  $n=2$ .  $n=2/\text{th}$  raíces de la unidad. Estos subproblemas tienen exactamente la misma forma que el problema original, pero tienen la mitad del tamaño. Ahora hemos dividido con éxito un cálculo  $DFTn$  de  $n$  elementos en dos cálculos  $DFTn=2$  de  $n=2$  elementos. Esta descomposición es la base del siguiente algoritmo FFT recursivo, que calcula la DFT de un vector de  $n$  elementos a  $D .a0; a1; \dots; an1/$ , donde  $n$  es una potencia de 2.

RECURSIVE-FFT.a/ 1 n

```

D a:longitud 2 si n ==
1 3
    devolver un
4 !n D e2 i=n 5 ! D 1
6 aŒ0 D .a0;
a2;:::;an2/ 7 aŒ1 D .a1; a3;:::;an1/
8 yŒ0 D RECURSIVE-FFT.aŒ0/ 9
yŒ1 D RECURSIVE-FFT.aŒ1/ 10 para k
D 0 a n=2 1 yk D yŒ0 C ! yŒ1 11 ykC.n=2/
D yŒ0 ! yŒ1

          k          k
12
13      ! D ! In 14          k
volver y                                // se supone que y es un vector columna

```

El procedimiento RECURSIVE-FFT funciona de la siguiente manera. Las líneas 2 y 3 representan la base de la recursividad; la DFT de un elemento es el propio elemento, ya que en este caso

```

y0 D a0 !0      1
D a0 1
D a0 :

```

Las líneas 6 y 7 definen los vectores de coeficientes para los polinomios AŒ0 y AŒ1. Las líneas 4, 5 y 13 garantizan que ! se actualiza correctamente para que siempre que se ejecuten las líneas 11 y 12, tengamos ! D ! k . (Manteniendo un valor de ejecución de ! desde la iteración desde cero cada a la iteración ahorra tiempo sobre la computación !k vez que pasa por for loop.) Las líneas 8–9 realizan los cálculos recursivos DFT<sub>n=2</sub>, estableciendo, para k D 0; 1; :: ; n=2 1,

```

yŒ0      D AŒ0.!k n=2/ ; D
yŒ1      AŒ1.!k n=2/ ;
o, ya que !k      D !2k por el lema de cancelación, n=2

```

```

yŒ0      D AŒ0.!2k / ; D
yŒ1      AŒ1.!2k / : _ 

```

Las líneas 11 y 12 combinan los resultados de los cálculos recursivos de DFTn=2 . Para y0; y1; ...; yn=21, la línea 11 produce

$$\begin{aligned} y_k &= y_0 e^{j \frac{2\pi}{2} k} + y_1 e^{-j \frac{2\pi}{2} k} \\ &= D A e^{j \frac{\pi}{2} k} / C e^{-j \frac{\pi}{2} k} n A e^{j \frac{\pi}{2} k} n / D \\ &= A e^{j \frac{\pi}{2} k} / \dots \quad (\text{por la ecuación (30.9)}). \end{aligned}$$

Para yn=2; yn=2C1; ...; yn1, siendo k D 0; 1; ...; n=2 1, la línea 12 produce

$$\begin{aligned} y_k &= y_0 e^{j \frac{2\pi}{2} k} + y_1 e^{-j \frac{2\pi}{2} k} \\ &= k C e^{j \frac{\pi}{2} k} C e^{-j \frac{\pi}{2} k} n A e^{j \frac{\pi}{2} k} n / D e^{j \frac{\pi}{2} k} \quad (\text{ya que } \frac{1}{2} C_n = \frac{1}{2} e^{j \frac{\pi}{2} k}) \\ &= D A e^{j \frac{\pi}{2} k} / C e^{-j \frac{\pi}{2} k} n A e^{j \frac{\pi}{2} k} n / \\ &= D A e^{j \frac{\pi}{2} k} C_n / C e^{-j \frac{\pi}{2} k} n A e^{j \frac{\pi}{2} k} n / \quad (\text{ya que } e^{j \frac{\pi}{2} k} C_n = e^{-j \frac{\pi}{2} k} n A e^{j \frac{\pi}{2} k} n) \\ &= D A e^{j \frac{\pi}{2} k} / \dots \quad (\text{por la ecuación (30.9)}). \end{aligned}$$

Por lo tanto, el vector y devuelto por RECURSIVE-FFT es de hecho el DFT del vector de entrada a.

Las líneas 11 y 12 multiplican cada valor  $y_k$  por  $e^{j \frac{2\pi}{2} k}$ , para  $k D 0; 1; \dots; n=2 1$ . y la línea 11 suma este producto al  $y_k$ , línea 12 lo resta. Porque usamos cada factor  $y_k$  en sus formas positiva y negativa, llamamos a los factores  $e^{j \frac{2\pi}{2} k}$  twiddle factores

Para determinar el tiempo de ejecución del procedimiento RECURSIVE-FFT, observamos que, excluyendo las llamadas recursivas, cada invocación toma un tiempo  $C_n$ , donde n es la longitud del vector de entrada. Por lo tanto, la recurrencia para el tiempo de ejecución es

$$T(n) = 2T(n/2) + C_n$$

Por lo tanto, podemos evaluar un polinomio de n ligado a grados en las raíces n-ésimas complejas de la unidad en el tiempo  $C_n \lg n$  usando la transformada rápida de Fourier.

### Interpolación en las raíces complejas de la unidad

Ahora completamos el esquema de multiplicación de polinomios mostrando cómo interpolar las raíces complejas de la unidad por un polinomio, lo que nos permite convertir de forma de valor puntual a forma de coeficiente. Interpolamos escribiendo la DFT como una ecuación matricial y luego observando la forma de la matriz inversa.

A partir de la ecuación (30.4), podemos escribir la DFT como el producto matricial  $y = Vn v$ , donde  $Vn$  es una matriz de Vandermonde que contiene las potencias apropiadas de  $n$ :

$y_0$	$1 \ 1 \ 1 \ 1 \ 2$				$a_0$
$y_1$	$1 \ 1 \ 1 \ 1 \ 2$		$1 \ 1 \ 3$	$1 \ 1 \ 1$	$a_1$
$y_2$	$1 \ 1 \ 1 \ 1 \ 2$	$1 \ 4$	$1 \ 6$	$1 \ 2 \cdot n \ 1 /$	$a_2$
$y_3$	$1 \ 1 \ 1 \ 1 \ 2$	$1 \ 6$	$1 \ 9$	$1 \ 3 \cdot n \ 1 /$	$a_3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$y_{n-1}$	$1 \ 1 \ 1 \ 1 \ 2$	$1 \ 2 \cdot n \ 1 /$	$1 \ 3 \cdot n \ 1 /$	$1 \ 1 \ 1 \cdot n \ 1 /$	$a_{n-1}$

La  $K; j$  / entrada de  $V_n$  es  $!k_j \mod n$ , para  $j; k \in 0; 1; \dots; n - 1$ . Los exponentes de la entradas de  $V_n$  forman una tabla de multiplicar.

Para la operación inversa, que escribimos como  $D \text{DFT}^{-1} \cdot y$ , procedemos por la inversa de  $V_n$ , multiplicando y por la matriz  $V$ .

### Teorema 30.7

para  $j; k \in 0; 1; \dots; n - 1$ , el  $j; k$  / entrada de  $V^{-1}$  es  $!k_j \mod n$ .

Prueba Mostramos que  $V^{-1} V_n = I_n$ , la matriz identidad  $n \times n$ . Considere el  $j; j$  / entrada de  $V^{-1} V_n$ :

$$\sum_{k=0}^{n-1} V_{nj} V_{nj}^* = \sum_{k=0}^{n-1} e^{j \frac{2\pi}{n} k j} = \sum_{k=0}^{n-1} e^{j \frac{2\pi}{n} k^2} = 1$$

$$\sum_{k=0}^{n-1} e^{j \frac{2\pi}{n} k^2} = \sum_{k=0}^{n-1} e^{j \frac{2\pi}{n} k^2} = 1$$

Esta sumatoria es igual a 1 si  $j = 0$ , y es 0 de lo contrario por el lema de suma  $j \neq 1$ , de modo que (Lema 30.6). Tenga en cuenta que confiamos en  $n \neq 1 / j$  no divisible por  $n$ , para que se aplique el lema de suma. ■

Dada la matriz inversa  $V^{-1}$ , tenemos que  $\text{DFT}^{-1} \cdot y$  viene dado por

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} x_{nk} y_k e^{-j \frac{2\pi}{n} k j} \quad (30.11)$$

para  $j \in 0; 1; \dots; n - 1$ . Al comparar las ecuaciones (30.8) y (30.11), vemos que al modificar el algoritmo FFT para cambiar los roles de  $a$  y  $y$ , reemplazar  $!n$  por  $!1$  y dividir cada elemento del resultado por  $n$ , calculamos el DFT inversa (vea el ejercicio 30.2-4). Por lo tanto, también podemos calcular DFT en  $n \lg n$  tiempo.

Vemos que, usando la FFT y la FFT inversa, podemos transformar un polinomio de grado  $n$  de ida y vuelta entre su representación de coeficiente y una representación de valor puntual en el tiempo  $n \lg n$ . En el contexto de la multiplicación de polinomios, hemos mostrado lo siguiente.

**Teorema 30.8 (Teorema de convolución)**

Para dos vectores cualesquiera  $a$  y  $b$  de longitud  $n$ , donde  $n$  es una potencia

de 2,  $a \circ b = DFT_1^{-1} [DFT_2n.a] \cdot DFT_2n.b]$  ;

donde los vectores  $a$  y  $b$  se rellenan con 0s hasta la longitud  $2n$  y denota el producto por componentes de dos vectores de  $2n$  elementos. ■

**Ejercicios****30.2-1**

Demostrar el Corolario 30.4.

**30.2-2**

Calcule la DFT del vector  $[0; 1; 2; 3]$ .

**30.2-3**

Resuelva el ejercicio 30.1-1 usando el esquema „ $n \lg n$ “-time.

**30.2-4**

Escriba un pseudocódigo para calcular  $DFT_1$  en „ $n \lg n$ “ tiempo.

**30.2-5**

Describa la generalización del procedimiento FFT para el caso en que  $n$  es una potencia de 3. Proporcione una recurrencia para el tiempo de ejecución y resuelva la recurrencia.

**30.2-6 ?**

Supongamos que en lugar de realizar una FFT de  $n$  elementos sobre el campo de números complejos (donde  $n$  es par), usamos el anillo  $Z_m$  de números enteros módulo  $m$ , donde  $m \geq 2^{\lceil \lg_2 n \rceil}$  y  $t$  es un número entero positivo arbitrario. Usar  $\sqrt[m]{t}$  en lugar de  $\sqrt[n]{t}$  como raíz  $n$ -ésima principal de la unidad, módulo  $m$ . Demuestre que la DFT y la DFT inversa están bien definidas en este sistema.

**30.2-7**

Dada una lista de valores  $[0; 1; \dots; n]$  (posiblemente con repeticiones), muestra cómo encontrar los coeficientes de un polinomio  $P(x)$  de grado  $n-1$  enlazado que tiene ceros solo en  $[0; 1; \dots; n]$  (posiblemente con repeticiones). Su procedimiento debería ejecutarse a tiempo en  $\lg_2 n$ . (Sugerencia: el polinomio  $P(x)$  tiene un cero en  $j$  si y solo si  $P(x)$  es un múltiplo de  $x-j$ .)

**30.2-8 ?**

La transformada chirp de un vector  $a = [a_0; a_1; \dots; a_n]$  es el vector  $y = [y_0; y_1; \dots; y_n]$ , donde  $y_k = \sum_{j=0}^{n-1} a_j e^{j \omega_k j}$  y  $\omega_k$  es cualquier número complejo. El

Por lo tanto, DFT es un caso especial de la transformada chirp, obtenida tomando  $D \ln n$ . Muestre cómo evaluar la transformada chirp en el tiempo en  $\lg n$  para cualquier número complejo'. (Sugerencia: utilice la ecuación

$$y_k D^k = \sum_{j=0}^{n-1} a_j e^{j \frac{2\pi}{n} j k}$$

para ver la transformación chirp como una convolución.)

### 30.3 Implementaciones eficientes de FFT

Dado que las aplicaciones prácticas de la DFT, como el procesamiento de señales, exigen la máxima velocidad, esta sección examina dos implementaciones eficientes de la FFT. Primero, examinaremos una versión iterativa del algoritmo FFT que se ejecuta en tiempo  $n \lg n$  pero puede tener una constante más baja oculta en la notación, que la versión recursiva de la Sección 30.2. (Dependiendo de la implementación exacta, la versión recursiva puede usar el caché de hardware de manera más eficiente). Luego, usaremos los conocimientos que nos llevaron a la implementación iterativa para diseñar un circuito FFT paralelo eficiente.

#### Una implementación iterativa de FFT

Primero notamos que el ciclo for de las líneas 10–13 de RECURSIVE-FFT involucra componiendo el valor  $y_k$  dos veces. En la terminología del compilador, llamamos a tal valor una subexpresión común. Podemos cambiar el ciclo para calcularlo solo una vez, almacenándolo en una variable temporal  $t$ .

para  $k$  de 0 a  $n-1$   $t$  re!

$$\begin{aligned} & y_0 \\ & \quad y_0 \\ & \quad \quad t \\ & \quad y_0 \\ & \quad \quad t \\ & \quad !D !Inote \end{aligned}$$

La operación en este ciclo, multiplicando el factor twiddle  $D^k$ , almacenar el producto en  $t$  y sumar y restar  $t$  de  $y_0$  se conoce como operación mariposa y se muestra esquemáticamente en la figura 30.3.

Ahora mostramos cómo hacer que la estructura del algoritmo FFT sea iterativa en lugar de recursiva. En la Figura 30.4, hemos dispuesto los vectores de entrada para las llamadas recursivas en una invocación de RECURSIVE-FFT en una estructura de árbol, donde la llamada inicial es para  $n=8$ . El árbol tiene un nodo para cada llamada del procedimiento, etiquetado

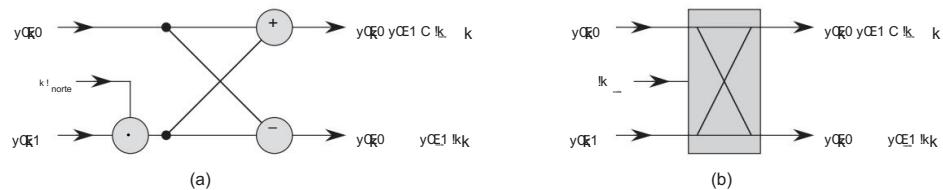


Figura 30.3 Una operación mariposa. (a) Los dos valores de entrada entran por la izquierda, el factor de giro  $k^{l-1}$  se multiplica por  $y[1]$ , y la suma y la diferencia se muestran a la derecha. (b) Un simplificado dibujo de una operación de mariposa. Usaremos esta representación en un circuito FFT paralelo.

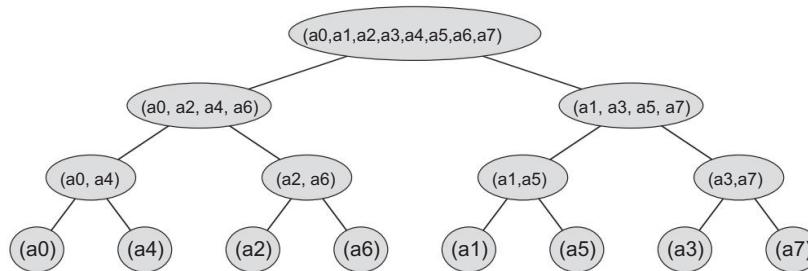


Figura 30.4 El árbol de vectores de entrada a las llamadas recursivas del procedimiento RECUSIVE-FFT . La invocación inicial es para  $n = 8$ .

por el vector de entrada correspondiente. Cada invocación RECUSIVE-FFT realiza dos llamadas recursivas, a menos que haya recibido un vector de 1 elemento. La primera llamada aparece en el niño izquierdo y la segunda llamada aparece en el niño derecho.

Mirando el árbol, observamos que si pudiéramos ordenar los elementos del vector inicial  $a$  en el orden en que aparecen en las hojas, podríamos rastrear la ejecución del procedimiento RECUSIVE-FFT, pero de abajo hacia arriba en lugar de arriba hacia abajo . Primero, tomamos los elementos en pares, calculamos la DFT de cada par usando una operación de mariposa y reemplazamos el par con su DFT. Luego, el vector contiene  $n = 2$  DFT de 2 elementos. Luego, tomamos estas  $n=2$  DFT en pares y calculamos la DFT de los cuatro elementos vectoriales de los que provienen ejecutando dos operaciones mariposa, reemplazando dos DFT de 2 elementos con una DFT de 4 elementos. Luego, el vector contiene  $n = 4$  DFT de 4 elementos. Continuamos de esta manera hasta que el vector contenga dos DFT de  $n=2$ /elementos, que combinamos usando  $n=2$  operaciones de mariposa en el DFT final de  $n$  elementos.

Para convertir este enfoque ascendente en código, usamos una matriz  $A \in \mathbb{C}^{n \times n}$  que inicialmente contiene los elementos del vector de entrada  $a$  en el orden en que aparecen

en las hojas del árbol de la figura 30.4. (Mostraremos más adelante cómo determinar este orden, que se conoce como permutación de inversión de bits). Debido a que tenemos que combinar DFT en cada nivel del árbol, introducimos una variable s para contar los niveles, que van desde 1 (en la parte inferior, cuando estamos combinando pares para formar DFT de 2 elementos) a  $\lg n$  (en la parte superior, cuando estamos combinando dos DFT de  $n=2^s$ -elemento para producir el resultado final). Por lo tanto, el algoritmo tiene la siguiente estructura:

```

1 para s D 1 a  $\lg n$  para
    k D 0 a n 1 por 2s 2 combine
    las dos DFT de 2s1 elementos en 3 k C 2s1 1 y AŒk C
    2s1 ::k C 2s 1      AŒk :: :
        en un elemento 2s DFT en AŒk :: k C 2s 1

```

Podemos expresar el cuerpo del ciclo (línea 3) como un pseudocódigo más preciso. Copiamos el ciclo for del procedimiento RECURSIVE-FFT, identificando yŒ0 con k C 2s1 1 e yŒ1 con AŒk C 2s1 ::k C 2s 1. El factor twiddle AŒk :: tor utilizado en cada operación mariposa depende del valor de s ; es una potencia de  $l_m$ , donde  $m \leq 2^s$ . (Introducimos la variable m únicamente en aras de la legibilidad).

Introducimos otra variable temporal u que nos permite realizar la operación mariposa en el lugar. Cuando reemplazamos la línea 3 de la estructura general por el cuerpo del bucle, obtenemos el siguiente pseudocódigo, que forma la base de la implementación en paralelo que presentaremos más adelante. El código primero llama al procedimiento auxiliar BIT-REVERSE-COPY.a; A/ para copiar el vector a en el arreglo A en el orden inicial en el que necesitamos los valores.

#### ITERATIVA-FFT.a/ 1

```

BIT-REVERSA-COPIA.a; A/ 2 n D
a:longitud // n es una potencia de 2 3 para s D 1 a  $\lg n$  4 m
D 2s !m D e2 i=m 5 6
para k D 0 a n 1
j      by m ! re 1 para
re 0 a m=2 1 t re ! AŒk C j C m=2 u D
    AŒk C
        j AŒk C j D u C t AŒk C
            j C m=2 D ut
7
8
9
10         ! D ! !metro
11 12 13 14 vuelta A

```

¿Cómo coloca BIT-REVERSE-COPY los elementos del vector de entrada a en el orden deseado en la matriz A? El orden en que aparecen las hojas en la figura 30.4

es una permutación de inversión de bits. Es decir, si dejamos que  $\text{rev.}k$  sea el entero de  $n$  bits lg formado al invertir los bits de la representación binaria de  $k$ , entonces queremos colocar el elemento vectorial  $a_k$  en la posición del arreglo  $A[\text{rev.}k]$ . En la Figura 30.4, por ejemplo, las hojas aparecen en el orden 0; 4; 2; 6; 1; 5; 3; 7; esta secuencia en binario es 000; 100; 010; 110; 001; 101; 011; 111, y cuando invertimos los bits de cada valor obtenemos la secuencia 000; 001; 010; 011; 100; 101; 110; 111. Para ver que queremos una permutación de inversión de bits en general, observamos que en el nivel superior del árbol, los índices cuyo bit de orden inferior es 0 van al subárbol izquierdo y los índices cuyo bit de orden inferior es 1 entran en el subárbol derecho. Eliminando el bit de orden inferior en cada nivel, continuamos este proceso hacia abajo en el árbol, hasta que obtenemos el orden dado por la permutación de inversión de bits en las hojas.

Dado que podemos calcular fácilmente la función  $\text{rev.}k$ , el procedimiento BIT-REVERSE-COPY es simple:

**BIT-REVERSA-COPIA.a; A/ 1**

```
n D a:longitud 2
para k D 0 a n 1
3      ACErev.k/ D ak
```

La implementación iterativa de FFT se ejecuta en el tiempo  $,n \lg n$ . La llamada a BIT REVERSE-COPY.a; A/ ciertamente se ejecuta en tiempo  $O(n \lg n)$ , ya que iteramos  $n$  veces y podemos invertir un número entero entre 0 y  $n-1$ , con  $\lg n$  bits, en tiempo  $O(\lg n)$ . (En la práctica, debido a que generalmente conocemos el valor inicial de  $n$  de antemano, probablemente codificaríamos una tabla que mapee  $k$  a  $\text{rev.}k$ , haciendo que BIT-REVERSE-COPY se ejecute en  $,n$  con una constante oculta baja. Alternativamente, podríamos usar el ingenioso esquema de contador binario inverso amortizado descrito en el problema 17-1.) Para completar la prueba de que la FFT ITERATIVA se ejecuta en el tiempo del bucle más interno (líneas 8 a 13) se ejecuta, es  $,n \lg n$ . El ciclo for de las líneas 6–13 itera  $n=m$  veces para cada valor de  $s$ , y el ciclo más interno de las líneas 8–13 itera  $m=2$  veces. De este modo,

Ln/DX	$\overbrace{\quad\quad}$ <small>node</small> 2s SD1	2s1
DX	$\overbrace{\quad\quad}$ <small>node</small> 2	SD1
$\lg n$		
$,n \lg n$ :		

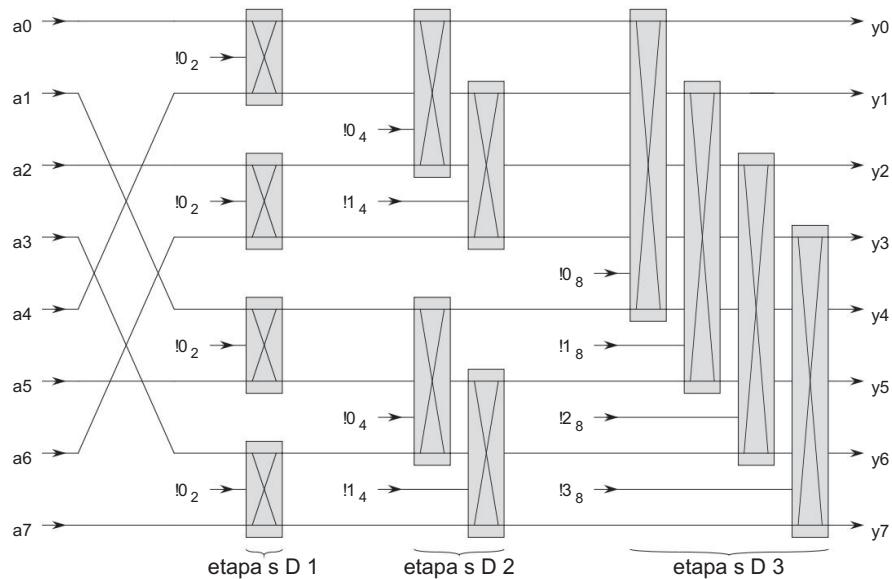


Figura 30.5 Un circuito que calcula la FFT en paralelo, aquí se muestra en  $n = 8$  entradas. Cada operación mariposa toma como entrada los valores de dos hilos, junto con un factor de giro, y produce como salida los valores de dos hilos. Las etapas de las mariposas están etiquetadas para corresponder a las iteraciones del ciclo más externo del procedimiento ITERATIVO-FFT . Solo los cables superior e inferior que pasan a través de una mariposa interactúan con ella; los cables que pasan por el medio de una mariposa no afectan a esa mariposa, ni sus valores cambian por esa mariposa. Por ejemplo, la mariposa superior en la etapa 2 no tiene nada que ver con el cable 1 (el cable cuya salida está etiquetada como  $y_1$ ); sus entradas y salidas están solo en los cables 0 y 2 (etiquetados como  $y_0$  y  $y_2$ , respectivamente). Este circuito tiene profundidad  $\lg n$  y realiza operaciones de mariposa  $\lg n$  en conjunto.

### Un circuito FFT paralelo

Podemos explotar muchas de las propiedades que nos permitieron implementar un algoritmo FFT iterativo eficiente para producir un algoritmo paralelo eficiente para la FFT. Expresaremos el algoritmo FFT paralelo como un circuito. La figura 30.5 muestra un circuito FFT en paralelo, que calcula la FFT en  $n$  entradas, para  $n = 8$ . El circuito comienza con una permutación inversa de bit de las entradas, seguida de  $\lg n$  etapas, cada etapa consta de  $n=2$  mariposas ejecutadas en paralelo. La profundidad del circuito, el número máximo de elementos computacionales entre cualquier salida y cualquier entrada que pueda alcanzarlo, es por lo tanto  $\lg n$ .

La parte más a la izquierda del circuito FFT paralelo realiza la permutación de bits inversos, y el resto imita el procedimiento iterativo ITERATIVO-FFT . Debido a que cada iteración del bucle for más externo realiza  $n = 2$  operaciones de mariposa independientes, el circuito las realiza en paralelo. El valor de  $s$  en cada iteración dentro de

ITERATIVE-FFT corresponde a una etapa de mariposas que se muestra en la Figura 30.5. Para  $s \leq D/2$ ; la etapa  $s$  consta de  $n=2^s$  grupos de mariposas (correspondientes a cada valor de  $k$  en ITERATIVE-FFT), con  $2^{s-1}$  mariposas por grupo (correspondientes a cada valor de  $j$  en ITERATIVE-FFT). Las mariposas que se muestran en la figura 30.5 corresponden a las operaciones de mariposa del ciclo más interno (líneas 9 a 12 de la FFT ITERATIVA). Nótese también que los factores de giro usados en las mariposas corresponden a los  $\omega^{k \cdot m}$ , usados en ITERATIVE-FFT: en la etapa  $s$ , usamos  $\omega^{k \cdot m}$  donde  $m = 2^s$ .

### Ejercicios

#### 30.3-1

Muestre cómo ITERATIVE-FFT calcula la DFT del vector de entrada  $[0; 2; 3; 1; 4; 5; 7; 9]$ .

#### 30.3-2

Muestre cómo implementar un algoritmo FFT con el anillo de permutación de inversión de bits al final, en lugar de al principio, del cálculo. (Sugerencia: Considere la DFT inversa.)

#### 30.3-3

¿Cuántas veces ITERATIVE-FFT calcula los factores de twiddle en cada etapa?

Vuelva a escribir ITERATIVE-FFT para calcular los factores de twiddle solo  $2^{s-1}$  veces en la etapa  $s$ .

#### 30.3-4

Suponga que los sumadores dentro de las operaciones mariposa del circuito FFT algunas veces fallan de tal manera que siempre producen una salida cero, independientemente de sus entradas. Suponga que exactamente un sumador ha fallado, pero no sabe cuál. Describa cómo puede identificar el sumador fallido suministrando entradas al circuito FFT general y observando las salidas. ¿Qué tan eficiente es su método?

### Problemas

a. Multiplicación divide y vencerás a. Muestre cómo multiplicar dos polinomios lineales  $ax + b$  y  $cx + d$  usando solo tres multiplicaciones. (Pista: una de las multiplicaciones es  $(a/c)x + (b/d)$ .)

b. Proporcione dos algoritmos de divide y vencerás para multiplicar dos polinomios de grado  $n$  enlazados en  $\lg n$  tiempo. El primer algoritmo debería dividir los coeficientes polinómicos de entrada en una mitad alta y una mitad baja, y el segundo algoritmo debería dividirlos según si su índice es par o impar.

C. Muestre cómo multiplicar dos enteros de  $n$  bits en  $O(n \lg 3)$  pasos, donde cada paso opera como máximo con un número constante de valores de 1 bit.

### 30-2 Matrices Toeplitz Una

matriz Toeplitz es una matriz  $nn \times nn$   $A$ .  $a_{ij} = a_{i+j}$  para  $i \in \{0, 1, 2, \dots, n-1\}$ ;  $j \in \{0, 1, 2, \dots, n-1\}$ .

a. ¿La suma de dos matrices Toeplitz es necesariamente Toeplitz? ¿Qué pasa con el producto?

b. Describa cómo representar una matriz Toeplitz para que pueda sumar dos matrices Toeplitz  $nn \times nn$  en  $O(n \lg n)$  tiempo.

C. Proporcione un algoritmo  $O(n \lg n)$  para multiplicar una matriz Toeplitz de  $nn \times nn$  por un vector de longitud  $n$ . Use su representación de la parte (b).

d. Proporcione un algoritmo eficiente para multiplicar dos matrices Toeplitz  $nn \times nn$ . Analiza su tiempo de ejecución.

### 30-3 Transformada rápida de Fourier multidimensional

Podemos generalizar la transformada discreta de Fourier unidimensional definida por la ecuación (30.8) a  $d$  dimensiones. La entrada es un arreglo  $d$ -dimensional  $AD$ .  $a_{j_1, j_2, \dots, j_d}$  cuyas dimensiones son  $n_1; n_2; \dots; n_d$ , donde  $n_1 n_2 \dots n_d = N$ . Definimos la transformada de Fourier discreta  $d$ -dimensional por la ecuación

$$y_{k_1, k_2, \dots, k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \dots \sum_{j_d=0}^{n_d-1} a_{j_1, j_2, \dots, j_d} e^{-j_1 k_1 \frac{2\pi}{n_1}} e^{-j_2 k_2 \frac{2\pi}{n_2}} \dots e^{-j_d k_d \frac{2\pi}{n_d}}$$

para  $0 \leq k_1 < n_1, 0 \leq k_2 < n_2, \dots, 0 \leq k_d < n_d$ .

a. Demuestre que podemos calcular una DFT  $d$ -dimensional calculando las DFT unidimensionales en cada dimensión por turno. Es decir, primero calculamos  $n=n_1$  DFT unidimensionales separadas a lo largo de la dimensión 1. Luego, usando el resultado de las DFT a lo largo de la dimensión 1 como entrada, calculamos  $n=n_2$  DFT unidimensionales separadas a lo largo de la dimensión 2. Usando este resultado como entrada, calculamos  $n=n_3$  DFT unidimensionales separadas a lo largo de la dimensión 3, y así sucesivamente, hasta la dimensión  $d$ .

b. Muestre que el orden de las dimensiones no importa, de modo que podamos calcular una DFT  $d$ -dimensional calculando las DFT unidimensionales en cualquier orden de las  $d$  dimensiones.

C. Muestre que si calculamos cada DFT unidimensional calculando la transformada rápida de Fourier, el tiempo total para calcular una DFT d-dimensional es  $O(n \lg n)$ , independiente de  $d$ .

30-4 Evaluación de todas las derivadas de un polinomio en un punto Dado un polinomio  $A(x)$  de grado  $n$ , definimos su  $t$ -ésima derivada por

$$\text{si } t = 0$$

$$\begin{aligned} \text{En } A(x) &= \sum_{j=0}^n a_j x^j \\ &= \sum_{j=0}^n a_j \frac{x^j - x_0^j}{x - x_0} x_0^j \end{aligned}$$

De la representación del coeficiente  $a_0; a_1; \dots; a_n$  de  $A(x)$  y un punto dado  $x_0$ , deseamos determinar  $A'(x_0)$  para  $t = 0; 1; \dots; n-1$

a. Dados los coeficientes  $b_0; b_1; \dots; b_n$  tal que

$$A(x) = \sum_{j=0}^n b_j x^j$$

muestre cómo calcular  $A'(x_0)$ , para  $t = 0; 1; \dots; n-1$ , en  $O(n \lg n)$  time.

b. Explique cómo encontrar  $b_0; b_1; \dots; b_n$  en  $O(n \lg n)$  time, dado  $A(x_0)$  para  $k = 0; 1; \dots; n-1$

C. Pruebalo

$$A(x_0) = \sum_{r=0}^n \sum_{j=0}^{k-1} \frac{x_0^r - x_0^j}{x_0^r - x_0^j} f(j)$$

donde  $f(j)$  es el  $j$ -ésimo término de  $f$

$$f(j) = \begin{cases} 1 & \text{si } j = 0 \\ 0 & \text{si } j \neq 0 \end{cases}$$

d. Explique cómo evaluar  $A(x_0)$  para  $k = 0; 1; \dots; n-1$  en  $O(n \lg n)$  time. Concluya que podemos evaluar todas las derivadas no triviales de  $A(x)$  en  $x_0$  en el tiempo  $O(n \lg n)$ .

### 30-5 Evaluación de polinomios en múltiples puntos

Hemos visto cómo evaluar un polinomio de grado  $n$  en un solo punto en  $O(n \cdot \text{time})$  usando la regla de Horner. También hemos descubierto cómo evaluar tal polinomio en todas las  $n$  raíces complejas de la unidad en  $O(n \lg n \cdot \text{time})$  usando la FFT. Ahora mostraremos cómo evaluar un polinomio de grado  $n$  en  $n$  puntos arbitrarios en  $O(n \lg 2^n \cdot \text{time})$ .

Para hacerlo, supondremos que podemos calcular el resto del polinomio cuando uno de esos polinomios se divide por otro en  $O(n \lg n \cdot \text{time})$ , resultado que establecemos sin demostración. Por ejemplo, el resto de  $3x^3 + Cx^2 + 3x + C_1$  cuando se divide por  $x^2 + Cx + C_2$  es

$$(3x^3 + Cx^2 + 3x + C_1) / (x^2 + Cx + C_2) \equiv 3x + C \pmod{x^2 + Cx + C_2}$$

Dada la representación del coeficiente de un polinomio  $A(x) = \sum_{k=0}^{n-1} a_k x^k$  y  $n$  puntos  $x_0, x_1, \dots, x_{n-1}$ , deseamos calcular los  $n$  valores  $A(x_0), A(x_1), \dots, A(x_{n-1})$ . Para  $0 \leq i \leq n-1$ , defina los polinomios  $P_{ij}(x) = A(x) - Q_{ij}(x)$ . Nótese que  $Q_{ij}(x)$  tiene grado como mucho  $j$ .

- a. Demostrar que  $A(x) \equiv Q(x) \pmod{A(x)}$  para cualquier punto  $x$ .
- b. Demuestre que  $Q(x) \equiv Q(x_0), Q(x_1), \dots, Q(x_{n-1}) \pmod{A(x)}$ .
- c. Pruebe que para  $0 \leq i < j \leq n-1$ , tenemos  $Q(x_i) \equiv Q(x_j) \pmod{P_{ij}(x)}$ .
- d. Proporcione un algoritmo  $O(n \lg 2^n \cdot \text{time})$  para evaluar  $A(x_0), A(x_1), \dots, A(x_{n-1})$ .

### 30-6 FFT usando aritmética modular

Como se define, la transformada discreta de Fourier requiere que calculemos con números complejos, lo que puede resultar en una pérdida de precisión debido a errores de redondeo. Para algunos problemas, se sabe que la respuesta contiene solo números enteros y, al usar una variante de la FFT basada en aritmética modular, podemos garantizar que la respuesta se calcule exactamente. Un ejemplo de tal problema es el de multiplicar dos polinomios con coeficientes enteros. El ejercicio 30.2-6 da un enfoque, usando un módulo de longitud  $\lfloor n/2 \rfloor$  bits para manejar una DFT en  $n$  puntos. Este problema da otro enfoque, que usa un módulo de la longitud más razonable  $O(\lg n)$ ; requiere que comprenda el material del Capítulo 31. Sea  $n$  una potencia de 2.

- a. Supongamos que buscamos el  $k$  más pequeño tal que  $p \mid kn - 1$  sea primo. Proporcione un argumento heurístico simple de por qué podríamos esperar que  $k$  sea aproximadamente  $\ln n$ . (El valor de  $k$  puede ser mucho mayor o menor, pero podemos esperar razonablemente examinar  $O(\lg n)$  valores candidatos de  $k$  en promedio). ¿Cómo se compara la longitud esperada de  $p$  con la longitud de  $n$ ?

Sea  $g$  un generador de  $Z_{\text{pag.}}$  y sea  $w \in D$  tal que  $w^k \equiv 1 \pmod p$ .

b. Argumente que la DFT y la DFT inversa son operaciones inversas módulo  $p$  bien definidas, donde  $w$  se usa como raíz enésima principal de la unidad.

C. Muestre cómo hacer que la FFT y su inversa funcionen módulo  $p$  en el tiempo en  $O(\lg n)$ , donde las operaciones en palabras de  $O(\lg n)$  bits toman la unidad de tiempo. Suponga que algoritmo se le da  $p$  y  $w$ .

d. Calcule el módulo DFT módulo  $p$  de los primeros 17 términos del vector  $\langle 0; 5; 3; 7; 7; 2; 1; 6 \rangle$ . Tenga en cuenta que  $g \in D_3$  es un generador de  $Z_{17}$

### Notas del capítulo

El libro de Van Loan [343] ofrece un excelente tratamiento de la transformada rápida de Fourier. Press, Teukolsky, Vetterling y Flannery [283, 284] tienen una buena descripción de la transformada rápida de Fourier y sus aplicaciones. Para una excelente introducción al procesamiento de señales, un área de aplicación popular de FFT, consulte los textos de Oppenheim y Schafer [266] y Oppenheim y Willsky [267]. El libro de Oppenheim y Schafer también muestra cómo manejar casos en los que  $n$  no es una potencia entera de 2.

El análisis de Fourier no se limita a datos unidimensionales. Es ampliamente utilizado en el procesamiento de imágenes para analizar datos en 2 o más dimensiones. Los libros de Gonzalez y Woods [146] y Pratt [281] analizan las transformadas multidimensionales de Fourier y su uso en el procesamiento de imágenes, y los libros de Tolimieri, An y Lu [338] y Van Loan [343] analizan las matemáticas de las transformadas multidimensionales rápidas de Fourier.

A Cooley y Tukey [76] se les atribuye ampliamente el diseño de la FFT en la década de 1960. De hecho, la FFT se había descubierto muchas veces anteriormente, pero su importancia no se comprendió por completo antes de la llegada de las computadoras digitales modernas. Aunque Press, Teukolsky, Vetterling y Flannery atribuyen los orígenes del método a Runge y König en 1924, un artículo de Heideman, Johnson y Burrus [163] rastrea la historia de la FFT desde CF Gauss en 1805.

Frigo y Johnson [117] desarrollaron una implementación rápida y flexible de la FFT, llamada FFTW ("la transformada de Fourier más rápida en Occidente"). FFTW está diseñado para situaciones que requieren múltiples cálculos de DFT en el mismo tamaño de problema. Antes de calcular realmente las DFT, FFTW ejecuta un "planificador" que, mediante una serie de ejecuciones de prueba, determina la mejor manera de descomponer el cálculo de FFT para el tamaño del problema dado en la máquina host. FFTW se adapta para usar el caché de hardware de manera eficiente y, una vez que los subproblemas son lo suficientemente pequeños, FFTW los resuelve con un código de línea recta optimizado. Además, FFTW tiene la ventaja inusual de tomar  $O(n \lg n)$  tiempo para cualquier tamaño de problema  $n$ , incluso cuando  $n$  es un número primo grande.

Aunque la transformada de Fourier estándar asume que la entrada representa puntos que están uniformemente espaciados en el dominio del tiempo, otras técnicas pueden aproximar la FFT en datos "no equiespaciados". El artículo de Ware [348] proporciona una descripción general.

---

## 31 Algoritmos de teoría de números

La teoría de números alguna vez fue vista como un tema hermoso pero en gran medida inútil en matemáticas puras. Hoy en día, los algoritmos de teoría de números se utilizan ampliamente, debido en gran parte a la invención de esquemas criptográficos basados en grandes números primos. Estos esquemas son factibles porque podemos encontrar números primos grandes fácilmente y son seguros porque no sabemos cómo factorizar el producto de números primos grandes (o resolver problemas relacionados, como calcular logaritmos discretos) de manera eficiente. Este capítulo presenta parte de la teoría de números y algoritmos relacionados que subyacen a tales aplicaciones.

La sección 31.1 introduce conceptos básicos de teoría de números, como divisibilidad, equivalencia modular y factorización única. La sección 31.2 estudia uno de los algoritmos más antiguos del mundo: el algoritmo de Euclides para calcular el máximo común divisor de dos enteros. La Sección 31.3 revisa conceptos de aritmética modular. La sección 31.4 luego estudia el conjunto de múltiplos de un número dado  $a$ , módulo  $n$ , y muestra cómo encontrar todas las soluciones a la ecuación  $ax \equiv b \pmod{n}$  usando el algoritmo de Euclides. El teorema chino del resto se presenta en la Sección 31.5. La sección 31.6 considera las potencias de un número dado  $a$ , módulo  $n$ , y presenta un algoritmo de cuadrados repetidos para calcular eficientemente  $a^b \pmod{n}$ , dados  $a$ ,  $b$  y  $n$ . Esta operación está en el corazón de las pruebas de primalidad eficientes y de gran parte de la criptografía moderna. La Sección 31.7 luego describe el criptosistema de clave pública RSA. La sección 31.8 examina una prueba de primalidad aleatoria. Podemos usar esta prueba para encontrar números primos grandes de manera eficiente, lo que debemos hacer para crear claves para el sistema criptográfico RSA. Finalmente, la Sección 31.9 revisa una heurística simple pero efectiva para factorizar números enteros pequeños. Es un hecho curioso que la factorización es un problema que la gente desearía ser intratable, ya que la seguridad de RSA depende de la dificultad de factorizar números enteros grandes.

### Tamaño de las entradas y costo de los cálculos aritméticos

Debido a que trabajaremos con números enteros grandes, debemos ajustar nuestra forma de pensar sobre el tamaño de una entrada y sobre el costo de las operaciones aritméticas elementales.

En este capítulo, una "entrada grande" generalmente significa una entrada que contiene "números enteros grandes" en lugar de una entrada que contiene "muchos números enteros" (como para ordenar). De este modo,

mediremos el tamaño de una entrada en términos de la cantidad de bits necesarios para representar esa entrada, no solo la cantidad de enteros en la entrada. Un algoritmo con entradas enteras  $a_1; a_2; \dots; a_k$  es un algoritmo de tiempo polinomial si se ejecuta en polinomio de tiempo en  $\lg a_1; \lg a_2; \dots; \lg a_k$ , es decir, polinomio en las longitudes de sus entradas codificadas en binario.

En la mayor parte de este libro, hemos encontrado conveniente pensar en las operaciones aritméticas elementales (multiplicaciones, divisiones o cálculo de residuos) como operaciones primitivas que toman una unidad de tiempo. Al contar el número de tales operaciones aritméticas que realiza un algoritmo, tenemos una base para hacer una estimación razonable del tiempo de ejecución real del algoritmo en una computadora. Sin embargo, las operaciones elementales pueden consumir mucho tiempo cuando sus insumos son grandes. Por lo tanto, se vuelve conveniente medir cuántas operaciones de bits requiere un algoritmo de teoría de números. En este modelo, la multiplicación de dos enteros de  $\lceil b \rceil$  por el método ordinario utiliza operaciones de  $\lceil b \rceil^2/\lceil b \rceil$ . De manera similar, podemos dividir un entero de  $\lceil b \rceil$  por un entero más corto o tomar el resto de un entero de  $\lceil b \rceil$  cuando se divide por un entero más corto en el tiempo  $\lceil b \rceil^2/\lceil b \rceil$  mediante algoritmos simples. (Consulte el ejercicio 31.1-12.) Se conocen métodos más rápidos. Por ejemplo, un método simple de dividir y vencerás para multiplicar dos enteros de  $\lceil b \rceil$  tiene un tiempo de ejecución de  $\lceil b \rceil \lg \lceil b \rceil$ , y el método más rápido conocido tiene un tiempo de ejecución de  $\lceil b \rceil \lg \lceil b \rceil \lg \lceil b \rceil$ . Sin embargo, para propósitos prácticos, el algoritmo  $\lceil b \rceil^2/\lceil b \rceil$  suele ser el mejor, y usaremos este límite como base para nuestros análisis.

En general, analizaremos los algoritmos en este capítulo en términos tanto del número de operaciones aritméticas y el número de operaciones de bit que requieren.

### 31.1 Nociones elementales de teoría de números

Esta sección proporciona una breve revisión de las nociones de la teoría elemental de números sobre el conjunto ZD f::: 2; 1; 0; 1; 2; : : : g de enteros y el conjunto ND f0; 1; 2; : : : g de números naturales.

#### Divisibilidad y divisores

La noción de que un número entero es divisible por otro es clave para la teoría de los números. La notación dja (léase "d divide a") significa que a D kd para algún entero k.

Todo número entero divide a 0. Si a>0 y dja, entonces jdj jaj. Si dja, entonces también decimos que a es múltiplo de d. Si d no divide a, escribimos d – a.

Si dja yd 0, decimos que d es divisor de a. Tenga en cuenta que dja si y solo si dja, por lo que no se pierde generalidad al definir los divisores como no negativos, entendiendo que el negativo de cualquier divisor de a también divide a a. A

divisor de un entero distinto de cero a es al menos 1 pero no mayor que  $\lfloor a \rfloor$ . Por ejemplo, los divisores de 24 son 1, 2, 3, 4, 6, 8, 12 y 24.

Todo entero positivo a es divisible por los divisores triviales 1 y a. Los divisores no triviales de a son los factores de a. Por ejemplo, los factores de 20 son 2, 4, 5 y 10.

### números primos y compuestos

Un número entero  $a > 1$  cuyos únicos divisores son los divisores triviales 1 y a es un número primo o, más simplemente, un número primo. Los números primos tienen muchas propiedades especiales y juegan un papel fundamental en la teoría de números. Los primeros 20 primos, en orden, son

2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71

El ejercicio 31.1-2 le pide que demuestre que hay infinitos números primos. Un número entero  $a > 1$  que no es primo es un número compuesto o, más simplemente, un número compuesto. Por ejemplo, 39 es compuesto porque 3 divide a 39. Al entero 1 lo llamamos unidad y no es ni primo ni compuesto. De manera similar, el entero 0 y todos los enteros negativos no son ni primos ni compuestos.

### El teorema de la división, los residuos y la equivalencia modular

Dado un entero n, podemos dividir los enteros en aquellos que son múltiplos de n y aquellos que no son múltiplos de n. Gran parte de la teoría de números se basa en refinar esta partición clasificando los no múltiplos de n según sus residuos cuando se dividen por n. El siguiente teorema proporciona la base para este refinamiento.

Omitimos la demostración (pero véase, por ejemplo, Niven y Zuckerman [265]).

#### Teorema 31.1 (Teorema de la división)

Para cualquier entero a y cualquier entero positivo n, existen enteros únicos q y r tales que  $0 \leq r < n$  y  $a = qn + r$ .

El valor  $q = a \text{ div } n$  es el cociente de la división. El valor  $r = a \text{ mod } n$  es el resto (o residuo) de la división. Tenemos ese resultado si y solo si  $a \equiv r \pmod{n}$ .

Podemos particionar los números enteros en n clases de equivalencia según su resto en los principales módulo n. La clase de equivalencia módulo n que contiene un entero a es

$\{x \in \mathbb{Z} : x \equiv a \pmod{n}\}$

Por ejemplo,  $\{x \in \mathbb{Z} : x \equiv 1 \pmod{3}\} = \{x \in \mathbb{Z} : x = 3k + 1 \text{ para algún } k \in \mathbb{Z}\}$ ; también podemos denotar este conjunto por  $\{x \in \mathbb{Z} : x \equiv 1 \pmod{3}\}$ . Usando la notación definida en la página 54, podemos decir que escribir  $\{x \in \mathbb{Z} : x \equiv 1 \pmod{3}\}$  es lo mismo que escribir  $\{x \in \mathbb{Z} : x \equiv 1 \pmod{3}\}$ . El conjunto de todas estas clases de equivalencia es

$Zn D \subset \{0, 1, \dots, n-1\}$  : (31.1)

Cuando ves la definición

$\forall n \in \mathbb{Z}^+$ ;  $\exists D \subseteq \mathbb{Z}^+$  ;  $\forall a \in \mathbb{Z}$ ;  $\exists b \in D$  ;  $a \equiv b \pmod{n}$ ; (31.2)

debe leerlo como equivalente a la ecuación (31.1) con el entendimiento de que 0 representa  $\{0\}$ , 1 representa  $\{1\}$ , y así sucesivamente; cada clase está representada por su elemento no negativo más pequeño. Sin embargo, debe tener en cuenta las clases de equivalencia subyacentes. Por ejemplo, si nos referimos a 1 como miembro de  $Z_n$ , en realidad nos estamos refiriendo a  $\{1\}$ , ya que  $1 \equiv 1 \pmod{n}$ .

### Divisores comunes y máximos comunes divisores

Si  $d$  es un divisor de  $a$  y  $d$  también es un divisor de  $b$ , entonces  $d$  es un divisor común de  $a$  y  $b$ . Por ejemplo, los divisores de 30 son 1, 2, 3, 5, 6, 10, 15 y 30, por lo que los divisores comunes de 24 y 30 son 1, 2, 3 y 6. Tenga en cuenta que 1 es un común divisor de dos enteros cualesquiera.

Una propiedad importante de los divisores comunes es que

$d|a$  y  $d|b$  implican  $d|a+b$  ; (31.3)

De manera más general, tenemos que

$d|a$  y  $d|b$  implica  $d|ax+by$  ; (31.4)

para cualquier número entero  $x$  e  $y$ . Además, si  $a|b$ , entonces  $a|b$  ;  $a|b$  o  $b=0$ , que  $a|b$  implica que

$a|b$  y  $b|a$  implican  $a=b$  ; (31.5)

El máximo común divisor de dos números enteros  $a$  y  $b$ , que no sean ambos cero, es el mayor de los divisores comunes de  $a$  y  $b$ ; lo denotamos por  $\gcd(a, b)$ . Por ejemplo,  $\gcd(24, 30) = 6$ ,  $\gcd(5, 7) = 1$  y  $\gcd(0, 9) = 9$ . Si  $a$  y  $b$  son distintos de cero, entonces  $\gcd(a, b)$  es un entero entre 1 y  $\min(a, b)$ . Definimos  $\gcd(0, 0) = 0$ ; esta definición es necesaria para que las propiedades estándar de la función gcd (como la ecuación (31.9) a continuación) sean universalmente válidas.

Las siguientes son propiedades elementales de la función mcd:

$\gcd(a, b) = \gcd(b, a)$  ;  $\gcd(a, b) = \gcd(b, a)$  ; (31.6)

$\gcd(a, b) = \gcd(b, a)$  ;  $\gcd(a, b) = \gcd(b, a)$  ; (31.7)

$\gcd(a, b) = \gcd(b, a)$  ;  $\gcd(a, b) = \gcd(b, a)$  ; (31.8)

(31.9)

para cualquier  $k \in \mathbb{Z}$  ; (31.10)

El siguiente teorema proporciona una caracterización alternativa y útil de  $\gcd(a, b)$ .

**Teorema 31.2**

Si  $a, b$  son enteros cualesquiera, no ambos cero, entonces  $\text{mcd}(a; b)$  es el elemento positivo más pequeño del conjunto  $\{ax + by \mid x, y \in \mathbb{Z}\}$  de combinaciones lineales de  $a$  y  $b$ .

**Demostración** Sea  $s$  la combinación lineal positiva más pequeña de  $a, b$ , y sea  $s = Dax + Cby$  para alguna  $x, y \in \mathbb{Z}$ . Sea  $q, r = sc$ . La ecuación (3.8) implica entonces

$$a \bmod s \equiv D \cdot a - qs \equiv D \cdot a - q \cdot ax$$

$$C \bmod s \equiv D \cdot a - .1 \cdot qx \equiv C$$

$$b \bmod s \equiv .qy \equiv ;$$

y por lo tanto un  $a \bmod s$  es una combinación lineal de  $a$  y  $b$  también. Pero, como  $0 \bmod s < s$ , tenemos que  $a \bmod s \neq 0$ , porque  $s$  es la combinación lineal positiva más pequeña. Por tanto, tenemos que  $s \mid a$  y, por razonamiento análogo,  $s \mid b$ .

Por lo tanto,  $s$  es un divisor común de  $a, b$ , por lo que  $\text{mcd}(a; b) \mid s$ . La ecuación (31.4) implica que  $\text{mcd}(a; b) \mid s$ , ya que  $\text{gcd}(a; b)$  divide tanto a como a  $b$  y  $s$  es una combinación lineal de  $a$  y  $b$ . Pero  $\text{gcd}(a; b) \mid s$  y  $s > 0$  implican que  $\text{gcd}(a; b) \mid s$ . Combinando  $\text{gcd}(a; b) \mid s$  y  $s \mid \text{gcd}(a; b)$  produce  $\text{gcd}(a; b) \mid s$ . Concluimos que  $s$  es el máximo común divisor de  $a$  y  $b$ . ■

**Corolario 31.3**

Para cualquier entero  $a, b$ , si  $d \mid a$  y  $d \mid b$ , entonces  $d \mid \text{gcd}(a; b)$ .

**Prueba** Este corolario se sigue de la ecuación (31.4), porque  $\text{mcd}(a; b) \mid s$  es una combinación lineal de  $a$  y  $b$  por el Teorema 31.2. ■

**Corolario 31.4**

Para todos los enteros  $a$  y  $b$  y cualquier entero no negativo  $n$ ,

$$\text{gcd}(an; bn) \mid D_n \text{ gcd}(a; b);$$

**Prueba** Si  $n = 0$ , el corolario es trivial. Si  $n > 0$ , entonces  $\text{gcd}(an; bn) \mid bn$  es el elemento positivo más pequeño del conjunto  $\{fanx + bny \mid x, y \in \mathbb{Z}\}$ , que es  $n$  veces el elemento positivo más pequeño del conjunto  $\{ax + by \mid x, y \in \mathbb{Z}\}$ . ■

**Corolario 31.5**

Para todos los enteros positivos  $n$ ,  $a, b$ , si  $n \mid ab$  y  $\text{gcd}(a; b) \mid D_1$ , luego  $n \mid b$ .

**Demostración** Dejamos la demostración como Ejercicio 31.1-5. ■

## Números enteros relativamente primos

Dos enteros  $a$  y  $b$  son primos relativos si su único divisor común es 1, es decir, si  $\text{mcd}(a; b) = 1$ . Por ejemplo, 8 y 15 son primos relativos, ya que los divisores de 8 son 1, 2, 4 y 8, y los divisores de 15 son 1, 3, 5 y 15. El siguiente teorema establece que si dos números enteros son primos relativos a un número  $p$ , entonces su producto es primo relativo a  $p$ .

## Teorema 31.6

Para cualesquiera enteros  $a$ ,  $b$  y  $p$ , si ambos  $\text{mcd}(a; p) = 1$  y  $\text{mcd}(b; p) = 1$ , luego  $\text{mcd}(ab; p) = 1$ .

Demostración Del Teorema 31.2 se sigue que existen los enteros  $x$ ,  $y$ ,  $x_0$  que nosotros, y tal

$$ax + bx_0 = 1 \quad : \quad \text{y}$$

$$ay + by_0 = 1 \quad : \quad \text{y}$$

Multiplicando estas ecuaciones y reordenando, tenemos

$$ab(xx_0 + by_0) = ab(1) = ab \quad : \quad \text{y}$$

Dado que 1 es una combinación lineal positiva de  $ab$  y  $p$ , una aplicación al Teorema 31.2 completa la demostración. ■

Los enteros  $n_1, n_2, \dots, n_k$  son pares relativamente primos si, siempre que  $i \neq j$ , tienen nosotros  
 $\text{mcd}(n_i; n_j) = 1$ ; Nueva Jersey / D 1.

## Factorización única

Un hecho elemental pero importante sobre la divisibilidad entre números primos es el siguiente.

## Teorema 31.7

Para todos los primos  $p$  y todos los enteros  $a$  y  $b$ , si  $p \mid ab$ , entonces  $p \mid a$  o  $p \mid b$  (o ambos).

Demostración Supóngase a los efectos de la contradicción que  $p \mid ab$ , pero que  $p \nmid a$  y  $p \nmid b$ . Así,  $\text{mcd}(a; p) = 1$  y  $\text{mcd}(b; p) = 1$ , ya que los únicos divisores de  $p$  son 1 y  $p$ , y suponemos que  $p$  no divide ni a ni b. El teorema 31.6 implica entonces que  $\text{mcd}(ab; p) = 1$ , contradiciendo nuestra suposición de que  $p \mid ab$ , ya que  $p \mid ab$  implica  $\text{mcd}(ab; p) > 1$ . Esta contradicción completa la prueba. ■

Una consecuencia del teorema 31.7 es que podemos factorizar de forma única cualquier compuesto entero en un producto de números primos.

Teorema 31.8 (Factorización única)

Hay exactamente una manera de escribir cualquier entero compuesto a como un producto de la forma

$a D p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$

donde los  $p_i$  son primos,  $p_1 < p_2 < \dots < p_r$ , y los  $e_i$  son números enteros positivos.

Demostración Dejamos la demostración como Ejercicio 31.1-11. ■

Como ejemplo, el número 6000 se factoriza únicamente en números primos como  $2^4 \cdot 3 \cdot 5^3$ .

### Ejercicios

31.1-1

Demuestre que si  $a > b > 0$  y  $c \mid a$   $c \mid b$ , entonces  $c \mid \text{mcd}(a, b)$ .

31.1-2

Demuestre que hay infinitos números primos. (Sugerencia: demuestre que ninguno de los primos  $p_1, p_2, \dots, p_k$  divide  $(p_1 p_2 \dots p_k)^2 + 1$ .)

31.1-3

Demuestre que si  $a \mid b$  y  $b \mid c$ , entonces  $a \mid c$ .

31.1-4

Demuestre que si  $p$  es primo y  $0 < k < p$ , entonces  $\text{mcd}(k, p) = 1$ .

31.1-5

Demuestre el Corolario 31.5.

31.1-6

Demuestre que si  $p$  es primo y  $0 < k < p$ , entonces  $p \mid a$  y  $p \mid b$   $\Rightarrow$   $\exists k$  tal que para todos los enteros  $n$ ,  $a \equiv b \pmod{p}$ .

31.1-7

Demuestre que si  $a$  y  $b$  son enteros positivos tales que  $a \mid b$ , entonces

$x \equiv a \pmod{b}$   $\Rightarrow$   $x \equiv a \pmod{a}$

para cualquier  $x$ . Demostrar, bajo los mismos supuestos, que

$x \equiv a \pmod{b} \Rightarrow x \equiv a \pmod{a}$

para cualquier número entero  $x$  e  $y$ .

## 31.1-8

Para cualquier entero  $k > 0$ , un entero  $n$  es una  $k$ -ésima potencia si existe un entero  $a$  tal que  $a^k \equiv n \pmod{D}$ . Además,  $n > 1$  es una potencia no trivial si es una  $k$ -ésima potencia para algún entero  $k > 1$ . Muestre cómo determinar si un entero  $n$  de  $\ell$ -bit dado es un polinomio de potencia en el tiempo no trivial en  $\ell$ .

## 31.1-9

Demuestre las ecuaciones (31.6)–(31.10).

## 31.1-10

Muestre que el operador  $\text{mcd}$  es asociativo. Es decir, demuestre que para todos los enteros  $a, b$  y  $c$ ,

$$\text{gcd}(a; \text{gcd}(b; c)) = \text{gcd}(\text{gcd}(a; b); c)$$

## 31.1-11 ?

Demuestre el teorema 31.8.

## 31.1-12

Proporcione algoritmos eficientes para las operaciones de dividir un entero de  $\ell$ -bit por un entero más corto y de tomar el resto de un entero de  $\ell$ -bit cuando se divide por un entero más corto. Sus algoritmos deberían ejecutarse en el tiempo  $\sim 2\ell$ .

## 31.1-13

Proporcione un algoritmo eficiente para convertir un número entero (binario) dado de  $\ell$ -bit a una representación decimal. Argumente que si la multiplicación o división de números enteros cuya longitud es como máximo  $\ell$  toma un tiempo  $M\ell$ , entonces podemos convertir binario a decimal en un tiempo  $\sim M\ell \lg \ell$ . (Sugerencia: use un enfoque de divide y vencerás, obteniendo las mitades superior e inferior del resultado con recursiones separadas).

## 31.2 Máximo común divisor

En esta sección, describimos el algoritmo de Euclides para calcular eficientemente el máximo común divisor de dos enteros. Cuando analicemos el tiempo de ejecución, veremos una conexión sorprendente con los números de Fibonacci, que producen una entrada en el peor de los casos para el algoritmo de Euclides.

Nos restringimos en esta sección a los enteros no negativos. Esta restricción es justificado por la ecuación (31.8), que establece que  $\text{mcd}(a; b) \mid \text{gcd}(ja; jb)$ .

En principio, podemos calcular  $\gcd(a; b)$  para enteros positivos  $a$  y  $b$  de las factorizaciones primas de  $a$  y  $b$ . De hecho, si

$$a = D p_1^{e_1} p_2^{e_2} \dots p_r^{e_r} \quad (31.11)$$

$$b = D p_1^{f_1} p_2^{f_2} \dots p_r^{f_r} \quad (31.12)$$

con exponentes cero que se utilizan para formar el conjunto de números primos  $p_1; p_2; \dots; p_r$  lo mismo para  $a$  y  $b$ , entonces, como se le pide que muestre en el Ejercicio 31.2-1,

$$\gcd(a; b) = D^{\min(e_1; f_1)} p_1^{\min(e_2; f_2)} \dots p_r^{\min(e_r; f_r)} \quad (31.13)$$

Sin embargo, como mostraremos en la sección 31.9, los mejores algoritmos hasta la fecha para la factorización no se ejecutan en tiempo polinomial. Por lo tanto, parece poco probable que este enfoque para calcular los máximos comunes divisores produzca un algoritmo eficiente.

El algoritmo de Euclides para calcular el máximo común divisor se basa en el siguiente teorema.

#### Teorema 31.9 (teorema de recursión GCD)

Para cualquier entero no negativo  $a$  y cualquier entero positivo  $b$ ,

$\gcd(a; b) = \gcd(b; a \bmod b)$ :

Prueba Demostraremos que  $\gcd(a; b) = \gcd(b; a \bmod b)$  se dividen entre sí, de modo que por la ecuación (31.5) deben ser iguales (ya que ambos son no negativos).

Primero mostramos que  $\gcd(a; b) \mid \gcd(b; a \bmod b)$  un modelo  $b$ . Si hacemos  $d \mid \gcd(a; b)$ , luego  $d \mid a$  y  $d \mid b$ . Por la ecuación (3.8),  $a \bmod b \mid b$ , donde  $q \mid a \bmod b$ .

Como  $a \bmod b$  es una combinación lineal de  $a$  y  $b$ , la ecuación (31.4) implica que  $d \mid a \bmod b$ . Por lo tanto, dado que  $d \mid b$  y  $d \mid a \bmod b$ , el Corolario 31.3 implica que  $d \mid \gcd(b; a \bmod b)$ , equivalentemente, que

$$\gcd(a; b) \mid \gcd(b; a \bmod b) \quad (31.14)$$

Mostrando que  $\gcd(b; a \bmod b) \mid \gcd(a; b)$  es casi lo mismo. Si ahora hacemos  $d \mid \gcd(b; a \bmod b)$ , luego  $d \mid b$  y  $d \mid a \bmod b$ . Como  $a \equiv a \bmod b \pmod{b}$ , donde  $q \mid a - a \bmod b$ , tenemos que  $a$  es una combinación lineal de  $b$  y  $a \bmod b$ . Por la ecuación (31.4), concluimos que  $d \mid a$ .

Desde  $d \mid b$  y  $d \mid a$ , tenemos que  $d \mid \gcd(a; b)$  por el Corolario 31.3 o, de manera equivalente, que

$$\gcd(b; a \bmod b) \mid \gcd(a; b) \quad (31.15)$$

El uso de la ecuación (31.5) para combinar las ecuaciones (31.14) y (31.15) completa la prueba. ■

### Algoritmo de Euclides

Los Elementos de Euclides (alrededor del 300 a. C.) describen el siguiente algoritmo gcd, aunque puede tener un origen incluso anterior. Expresamos el algoritmo de Euclides como un programa recursivo basado directamente en el Teorema 31.9. Las entradas  $a$  y  $b$  son números enteros no negativos arbitrarios.

```

EUCLID.a; b/ 1
si b == 0 2
devuelve a
3 más devuelve EUCLID.b; un modelo b/

```

Como ejemplo de la ejecución de EUCLID, considere el cálculo de gcd.30; 21/:

```

EUCLID.30; 21/D EUCLID.21; 9/D EUCLID.9;
    3/ D EUCLID.3; 0/ D
        3

```

Este cálculo llama a EUCLID recursivamente tres veces.

La corrección de EUCLID se deriva del Teorema 31.9 y de la propiedad de que si el algoritmo devuelve  $a$  en la línea 2, entonces  $b \leq 0$ , por lo que la ecuación (31.9) implica que  $\text{mcd}(a; b) \geq \text{mcd}(a; 0) = a$ . El algoritmo no puede recurrir indefinidamente, ya que el segundo argumento decrece estrictamente en cada llamada recursiva y siempre es no negativo.

Por lo tanto, EUCLID siempre termina con la respuesta correcta.

### El tiempo de ejecución del algoritmo de Euclides

Analizamos el tiempo de ejecución del peor de los casos de EUCLID en función del tamaño de  $a$  y  $b$ . Asumimos sin pérdida de generalidad que  $a > b \geq 0$ . Para justificar esta suposición, observe que si  $b > a$ , entonces EUCLID.a; b/ inmediatamente hace la llamada recursiva EUCLID.b; a/. Es decir, si el primer argumento es menor que el segundo, EUCLID gasta una llamada recursiva intercambiando sus argumentos y luego continúa. De manera similar, si  $b < a > 0$ , el procedimiento termina después de una llamada recursiva, ya que  $a \bmod b = 0$ .

El tiempo de ejecución general de EUCLID es proporcional al número de llamadas recursivas que realiza. Nuestro análisis hace uso de los números de Fibonacci  $F_k$ , definidos por la recurrencia (3.22).

Lema 31.10 Si

$a > b > 1$  y la llamada EUCLID.a; b/ realiza  $k$  llamadas recursivas, luego  $a \bmod b = F_{k+1}$ .

Prueba La prueba procede por inducción sobre  $k$ . Para la base de la inducción, sea  $k \leq 1$ . Entonces,  $b \leq D F_2$ , y como  $a > b$ , debemos tener  $a \leq D F_3$ . Como  $b > a \bmod b$ , en cada llamada recursiva el primer argumento es estrictamente mayor que el segundo; la suposición de que  $a > b$  por lo tanto se cumple para cada llamada recursiva.

Asumo inductivamente que el lema se cumple si  $k \leq 1$  se realizan llamadas recursivas; luego probaremos que el lema se cumple para  $k \geq 2$  llamadas recursivas. Como  $k > 1$ , tenemos  $b > 0$  y  $\text{EUCLID}.a; b /$  llama a  $\text{EUCLID}.b; a \bmod b /$  recursivamente, que a su vez hace  $k - 1$  llamadas recursivas. La hipótesis inductiva entonces implica que  $b \leq F_{k-1}$  (probando así parte del lema), y  $a \bmod b \leq F_k$ . Tenemos

$$b \leq C \cdot a \bmod b / D b \leq C \cdot a \bmod b /$$

$\vdots$

ya que  $a > b > 0$  implica  $a \bmod b = b$ . Por lo tanto,

$$a \leq b \leq C \cdot a \bmod b /$$

$$F_{k-1} \leq F_k$$

D  $F_{k-1}$  :

■

El siguiente teorema es un corolario inmediato de este lema.

#### Teorema 31.11 (Teorema de Lame)

- ‘ Para cualquier entero  $k \geq 1$ , si  $a > b \geq 1$  y  $b < F_{k-1}$ , entonces el llamado  $\text{EUCLID}.a; b /$  realiza menos de  $k$  llamadas recursivas. ■

Podemos demostrar que la cota superior del teorema 31.11 es la mejor posible demostrando que la llamada  $\text{EUCLID}.F_k$ ;  $F_k /$  hace exactamente  $k$  llamadas recursivas cuando  $k \geq 2$ . Usamos inducción en  $k$ . Para el caso base,  $k = 2$ , y la llamada  $\text{EUCLID}.F_2$ ;  $F_2 /$  realiza exactamente una llamada recursiva a  $\text{EUCLID}.1; 0 /$ . (Tenemos que empezar en  $k = 2$ , porque cuando  $k = 1$  no tenemos  $F_2 > F_1$ .) Para el paso inductivo, suponga que  $\text{EUCLID}.F_k$ ;  $F_k /$  hace exactamente  $k$  llamadas recursivas. Para  $k > 2$ , tenemos  $F_k > F_{k-1} > 0$  y  $F_k \leq D F_k$  ( $F_k \leq F_{k-1} + 1$ ), y así por el ejercicio 31.1-1, tenemos  $F_k \leq F_{k-1} + 1$ . Así, tenemos

$$\text{gcd}.F_k; F_k / \leq \text{gcd}.F_{k-1}; F_{k-1} / + 1$$

$\vdots$

Por lo tanto, la llamada  $\text{EUCLID}.F_k$ ;  $F_k /$  recurre una vez más que la llamada  $\text{EUCLID}.F_{k-1}$ ;  $F_{k-1} /$ , o exactamente  $k - 1$  veces, alcanzando el límite superior del Teorema 31.11.

Dado que  $F_k$  es aproximadamente  $k = \sqrt[3]{5}$ , donde la proporción áurea es  $\phi = \frac{\sqrt{5}+1}{2}$  definida por la ecuación (3.24), el número de llamadas recursivas en  $\text{EUCLID}$  es  $O(\lg b)$ . (Ver

a	b	ba = bc	d	x	y
99	78		1	3	11
78	21	21	3	3	3
15	15	6	1	3	2
2	6	3	3	1	2
			0		
			1		10

Figura 31.1 Cómo calcula EXTENDED-EUCLID gcd.99; 78/. Cada línea muestra un nivel de la recursividad: los valores de las entradas a y b, el valor calculado ba=bc y los valores d, x e y devueltos. El triple .d; X; y/ devuelto se convierte en el triple .d0 ; x0 ; y0 / utilizado en el siguiente nivel superior de recursividad. La llamada EXTENDIDO-EUCLID.99; 78/ devuelve .3; 11; 14/, por lo que mcd.99; 78/ D 3 D 99 .11/ C 78 14.

Ejercicio 31.2-5 para un límite más estrecho). Por lo tanto, si llamamos a EUCLID en dos números de ~bit, entonces realiza O.^2/ operaciones aritméticas y O.^3/ operaciones de bit (suponiendo que la multiplicación y división de números de ~bit tomar operaciones de O.^2/ bit). El problema 31-2 le pide que muestre un límite de O.^2/ en el número de operaciones de bits.

La forma extendida del algoritmo de Euclides

Ahora reescribimos el algoritmo de Euclides para calcular información útil adicional. Específicamente, extendemos el algoritmo para calcular los coeficientes enteros x e y tales que

d D gcd.a; b/ D hacha C por : (31.16)

Tenga en cuenta que x e y pueden ser cero o negativos. Veremos que estos coeficientes son útiles más adelante para calcular inversos multiplicativos modulares. El procedimiento EUCLID EXTENDIDO toma como entrada un par de enteros no negativos y devuelve un triple de la forma .d; X; y/ que satisface la ecuación (31.16).

```

EXTENDIDO-EUCLID.a; b/ 1
si b == 0 2
devuelve .a; 1; 0/ 3
más .d0 ; x0 ; y0 / D EXTENDIDO-EUCLID.b; un modelo b/ 4 .d;
X; y/ D .d0 ; y0 ; x0 ba=bc y0 / 5 devuelve .d; X;
tu/

```

La Figura 31.1 ilustra cómo EXTENDED-EUCLID calcula gcd.99; 78/.

El procedimiento EUCLID EXTENDIDO es una variación del procedimiento EUCLID . La línea 1 es equivalente a la prueba "b == 0" en la línea 1 de EUCLID. Si b D 0, entonces

EXTENDED-EUCLID devuelve no solo d D a en la línea 2, sino también los coeficientes x D 1 e y D 0, de modo que a D ax C by. Si b ≠ 0, EXTENDED-EUCLID primero calcula .d0 ; x0 ; y0 / tal que d0 D gcd.b; un modelo b/ y

d0 D bx0 C .a mod b/y0 : (31.17)

En cuanto a EUCLID, tenemos en este caso d D gcd.a; b/ D d0 D gcd.b; un modelo b/.

Para obtener x e y tales que d D ax C by, comenzamos reescribiendo la ecuación (31.17) usando la ecuación d D d0 y la ecuación (3.8):

d D bx0 C .ab ba=bc/y0

Día0 C b.x0 ba=bc y0 / :

Por lo tanto, elegir x D y0 y y D x0 ba=bc y0 satisface la ecuación d D ax C by, demostrando la corrección de EUCLID EXTENDIDA.

Dado que la cantidad de llamadas recursivas realizadas en EUCLID es igual a la cantidad de llamadas recursivas realizadas en EXTENDED-EUCLID, los tiempos de ejecución de EUCLID y EXTENDED-EUCLID son los mismos, dentro de un factor constante. Es decir, para a>b>0, el número de llamadas recursivas es O.lg b/.

### Ejercicios

31.2-1

Demuestre que las ecuaciones (31.11) y (31.12) implican la ecuación (31.13).

31.2-2

Calcule los valores .d; X; y/ que la llamada EXTENDIDO-EUCLID.899; 493/ devoluciones.

31.2-3

Demuestre que para todos los enteros a, k y n,

gcd.a; n/ D gcd.a C kn; n/ :

31.2-4

Reescriba EUCLID en una forma iterativa que use solo una cantidad constante de memoria (es decir, almacene solo una cantidad constante de valores enteros).

31.2-5

Si a>b 0, demuestre que la llamada EUCLID.a; b/ realiza como máximo 1 C log b llamadas recursivas. Mejore este límite a 1 C log.b= gcd.a; b//.

31.2-6

¿Qué significa EXTENDED-EUCLID.FkC1; Fk/ regresar? Demuestra que tu respuesta es correcta.

## 31.2-7

Defina la función gcd para más de dos argumentos mediante la ecuación recursiva  $\text{gcd}.\text{a}_0; \text{a}_1; \dots; \text{a}_n / \text{D gcd}.\text{a}_0; \text{gcd}.\text{a}_1; \text{a}_2; \dots; \text{a}_n //$ . Demuestre que la función mcd devuelve la misma respuesta independientemente del orden en que se especifican sus argumentos. También muestre cómo encontrar números enteros  $x_0; x_1; \dots; x_n$  tal que  $\text{gcd}.\text{a}_0; \text{a}_1; \dots; \text{a}_n / \text{D a}_0x_0 C a_1x_1 CC \dots C a_nx_n$ . Demuestre que el número de divisiones realizadas por su algoritmo es  $O(n \lg \max \{a_0; a_1; \dots; a_n\})$ .

## 31.2-8

Definir  $\text{lcm}.\text{a}_1; \text{a}_2; \dots; \text{a}_n /$  ser el mínimo común múltiplo de los  $n$  enteros  $a_1; a_2; \dots; a_n$ , es decir, el entero no negativo más pequeño que es un múltiplo de cada  $a_i$ .

Muestre cómo calcular  $\text{lcm}.\text{a}_1; \text{a}_2; \dots; \text{a}_n /$  usando eficientemente la operación gcd (de dos argumentos) como una subrutina.

## 31.2-9

Demuestre que  $n_1, n_2, n_3$  y  $n_4$  son pares relativos si y sólo si  $\text{mcd}.\text{n}_1\text{n}_2; \text{n}_3\text{n}_4 / \text{D gcd}.\text{n}_1\text{n}_3; \text{n}_2\text{n}_4 / \text{D } 1$

De manera más general, demuestre que  $n_1; n_2; \dots; n_k$  son pares relativamente primos si y solo si un conjunto de  $\text{dmg}$  ke pares de números derivados de ni son relativamente primos.

### 31.3 Aritmética modular

Informalmente, podemos pensar en la aritmética modular como la aritmética habitual sobre los números enteros, excepto que si estamos trabajando en el módulo  $n$ , entonces cada resultado  $x$  se reemplaza por el elemento de  $f_0; 1; \dots; n-1$  que es equivalente a  $x$ , módulo  $n$  (es decir,  $x$  se reemplaza por  $x \bmod n$ ). Este modelo informal es suficiente si nos atenemos a las operaciones de suma, resta y multiplicación. Un modelo más formal para la aritmética modular, que presentamos ahora, se describe mejor dentro del marco de la teoría de grupos.

#### Grupos finitos

Un grupo  $(S, \circ)$  es un conjunto  $S$  junto con una operación binaria  $\circ$  definida en  $S$  para la cual se cumplen las siguientes propiedades:

1. Cierre: Para todo  $a, b \in S$ , tenemos  $a \circ b \in S$ .
2. Identidad: Existe un elemento  $e \in S$ , llamado identidad del grupo, tal que  $e \circ a = a \circ e = a$  para todo  $a \in S$ .
3. Asociatividad: Para todo  $a, b, c \in S$ , tenemos  $(a \circ b) \circ c = a \circ (b \circ c)$ .

4. Inversos: Para cada  $a \in S$ , existe un único elemento  $b \in S$ , llamado el inverso de  $a$ , tal que  $a \circ b \in D$  y  $a \circ b = e$ .

Como ejemplo, considere el grupo familiar  $(\mathbb{Z}; +)$  de los enteros  $\mathbb{Z}$  bajo la operación de suma: 0 es la identidad, y el inverso de  $a$  es  $-a$ . Si un grupo  $(S; \circ)$  satisface la ley conmutativa  $a \circ b = b \circ a$  para todo  $a, b \in S$ , entonces es un grupo abeliano. Si un grupo  $(S; \circ)$  satisface  $|S| < 1$ , entonces es un grupo finito.

Los grupos definidos por suma y multiplicación modular

Podemos formar dos grupos abelianos finitos usando el módulo  $n$  de suma y multiplicación, donde  $n$  es un número entero positivo. Estos grupos se basan en las clases de equivalencia de los enteros módulo  $n$ , definidas en la Sección 31.1.

Para definir un grupo en  $\mathbb{Z}_n$ , necesitamos tener operaciones binarias adecuadas, que obtenemos al redefinir las operaciones ordinarias de suma y multiplicación.

Podemos definir fácilmente operaciones de suma y multiplicación para  $\mathbb{Z}_n$ , porque la clase de equivalencia de dos números enteros determina de manera única la clase de equivalencia de su suma o producto. Es decir, si  $a \equiv a_0 \pmod{n}$  y  $b \equiv b_0 \pmod{n}$ , entonces

$$\begin{aligned} a \circ b &\equiv a_0 + b_0 \pmod{n} ; \\ a \circ b &\equiv a_0 \cdot b_0 \pmod{n} ; \end{aligned}$$

Así, definimos el módulo  $n$  de suma y multiplicación, denotado  $\mathbb{Z}_n$  y  $\mathbb{Z}_n$ , por

$$\begin{aligned} &\text{Definición } \mathbb{Z}_n \text{ y } \mathbb{Z}_n \text{ ;} & (31.18) \\ &\text{Definición } n \text{ y } \mathbb{Z}_n \text{ ;} \end{aligned}$$

(Podemos definir la resta de  $\mathbb{Z}_n$  de manera similar por  $a - b \equiv a + (-b) \pmod{n}$ , pero la división es más complicada, como veremos). Estos hechos justifican la práctica común y conveniente de usar el elemento no negativo más pequeño de cada clase de equivalencia como su representante al realizar cálculos en  $\mathbb{Z}_n$ . Sumamos, restamos y multiplicamos como de costumbre sobre los representantes, pero reemplazamos cada resultado  $x$  por el representante de su clase, es decir, por  $x \pmod{n}$ .

Usando esta definición de suma módulo  $n$ , definimos el grupo aditivo módulo  $n$  como  $(\mathbb{Z}_n; +)$ . El tamaño del grupo aditivo módulo  $n$  es  $|\mathbb{Z}_n| = n$ .

La figura 31.2(a) da la tabla de operaciones para el grupo  $(\mathbb{Z}_6; +)$ .

### Teorema 31.12

El sistema  $(\mathbb{Z}_n; \cdot)$  es un grupo abeliano finito.

La ecuación de prueba (31.18) muestra que  $(\mathbb{Z}_n; \cdot)$  está cerrado. La asociatividad y conmutatividad de  $\mathbb{Z}_n$  se derivan de la asociatividad y conmutatividad de  $\mathbb{C}$ :

+6	012345
0	012345
1	12345 0
2	2345 0 1
3	345 012
4	4 5 0123
5	5 01234

(a)

15	1 2 4 7 8 11 13 14
1	1 2 4 7 8 11 13 14
2	2 4 8 14 1 7 11 13
4	4 8 1 13 2 14 7 11
7	7 14 13 4 11 2 1
8	8 1 2 11 4 13 14 7
11	11 7 14 2 13 1 8 4
13	13 11 7 1 14 8 4 2
14	14 13 11 8 7 4 2 1

(b)

Figura 31.2 Dos grupos finitos. Las clases de equivalencia se denotan por sus elementos representativos. (a) El grupo  $Z_6$ ;  $C_6$ . (b) El grupo  $Z_{15}$ .  $\langle 15 \rangle$

.Œan Cn Œbn/ Cn Œcn D Œa C bn Cn Œcn D Œ.a C b/ C cn D Œa  
C .b C c/n D Œan Cn Œb C  
en D Œan Cn Œb C Œan/

Œan Cn Œbn D Œa C bn

D œb C an

D OFbn Cn OFan

El elemento de identidad de  $\langle \text{Zn}; \text{Cn} \rangle$  es 0 (es decir,  $\text{COn}$ ). El inverso (aditivo) de un elemento  $a$  (es decir, de  $\langle \text{CAn} \rangle$ ) es el elemento  $a$  (es decir,  $\langle \text{CAn} \rangle$  o  $\langle \text{CEn} \rangle$ ) ya que  $\langle \text{CEn} \rangle \text{Cn} \langle \text{CEn} \rangle = \langle \text{D} \rangle$  ( $\text{CEn} \rangle \text{an} \langle \text{D} \rangle = \langle \text{CEn} \rangle$ )  $\text{CEn} \rangle = 0$ .

Usando la definición de multiplicación módulo  $n$ , definimos el grupo multiplicativo módulo  $n$  como  $Z_n^*$  que es el grupo de los elementos de  $Z_n$  que tienen un elemento inverso.

Z D f0Ean 2 Zn W qcd.a; n/ D 1g ;

Para ver que  $Z_{\text{mod } n}$  está bien definida, tenga en cuenta que para  $0 \leq a < n$ , tenemos  $a \in C_{kn}$ .  
 Un  $\text{mod } n$  para todos los enteros  $k$ . Por el ejercicio 31.2-3, por lo tanto,  $\text{gcd}(a; n) \mid D$  implica  $\text{mcd}(a; C) = kn$ ;  $n \mid D$  para todos los enteros  $k$ . Como  $\text{C} \cap D = \{0\}$ , el conjunto  $Z$  está bien definido.  
 Un ejemplo de tal grupo es

$\exists_{15} \text{ re } f1; 2; 4; 7; 8; 11; 13; 14 \text{ g;}$

donde la operación de grupo es la multiplicación módulo 15. (Aquí denotamos un elemento  $\text{C} \in \mathbb{Z}_{15}$  como  $a$ ; por ejemplo, denotamos  $\text{C} \in \mathbb{Z}_{15}$  como 7). La figura 31.2(b) muestra el grupo  $\mathbb{Z}_{13} \bmod 15$ , trabajando en  $\mathbb{Z}_{15}$  la identidad de este grupo es, por ejemplo, 8 11

15.

### Teorema 31.13

El sistema  $\mathbb{Z}_{n \bmod n}$  es un grupo abeliano finito.

Demostración El teorema 31.6 implica que  $\mathbb{Z}_{n \bmod n}$  está cerrado. Asociatividad y comun puede probarse para  $C_n$  en la demostración del teorema 31.12.

El elemento de identidad es  $\text{C} \in \mathbb{Z}_n$ . Para mostrar la existencia de inversas, sea  $a$  un elemento de  $\mathbb{Z}_n$  y sea  $d; X; y$  ser devuelto por EXTENDED-EUCLID. $a; n$ . Entonces,  $d \in \mathbb{Z}_1$ , desde un  $2 \in \mathbb{Z}_{n \bmod n}$ , y

hacha C y D 1 (31.19)

o equivalente,

hacha  $1 \bmod n$  :

Así,  $\text{C} \in \mathbb{Z}_n$  es un inverso multiplicativo de  $\text{C} \in \mathbb{Z}_n$ , módulo  $n$ . Además, afirmamos que  $\text{C} \in \mathbb{Z}_n$  para ver por qué, la ecuación (31.19) demuestra que la combinación lineal positiva más pequeña de  $x$  y  $n$  debe ser 1. Por lo tanto, el teorema 31.2 implica que  $\text{mcd}(x; n) = 1$ . Aplazamos la prueba de que las inversas están definidas de forma única hasta el Corolario 31.26.

Como ejemplo de cálculo de inversos multiplicativos, suponga que  $a \in \mathbb{Z}_5$  y  $n \in \mathbb{Z}_{11}$ . Entonces EXTENDED-EUCLID. $a; n$  devuelve  $d; X; y$  de modo que  $1 \in \mathbb{Z}_5 \cdot d / \mathbb{Z}_{11} \cdot 1$ . Por lo tanto,  $\text{C} \in \mathbb{Z}_{11}$  (es decir,  $\text{C} \in \mathbb{Z}_{11}$ ) es el inverso multiplicativo de  $\text{C} \in \mathbb{Z}_5$ .

En el resto de este capítulo, seguimos la práctica conveniente de denotar Al trabajar con los grupos  $\mathbb{Z}_n$ ;  $C_n$  y  $\mathbb{Z}_n$  las clases de equivalencia por sus elementos representativos y denotar las operaciones  $C_n$  y  $n$  por las notaciones aritméticas usuales  $C$  y ( $o$  juxtaposición, de modo que  $ab$  es  $ab$ ) respectivamente. Además, las equivalencias módulo  $n$  también pueden interpretarse como ecuaciones en  $\mathbb{Z}_n$ . Por ejemplo, las siguientes dos sentencias son equivalentes:

$\text{hacha } b \bmod n ; \text{C} \in \mathbb{Z}_n$   
 $n \text{C} \in \mathbb{Z}_n \text{D} \text{C} \in \mathbb{Z}_n$

Para mayor comodidad, a veces nos referimos a un grupo  $S$ ;  $^*$  / simplemente como  $S$  cuando la operación  $^*$  se entiende por contexto. Así podemos referirnos a los grupos  $\mathbb{Z}_n$ ;  $C_n$  y  $\mathbb{Z}_n$  como  $\mathbb{Z}_n$  y  $Z$  respectivamente.

Denotamos el inverso (multiplicativo) de un elemento  $a$  por  $a^{-1} \bmod n$ . La división se define por la ecuación en  $\mathbb{Z}_n$   $a=b \cdot ab^{-1} \bmod n$ . Por ejemplo, en  $\mathbb{Z}_5$

tenemos que  $7 \cdot 13 \equiv 1 \pmod{15}$ , ya que  $7 \cdot 13 \cdot 91 = 7 \cdot 13 \cdot 1 \pmod{15}$ , por lo que  $7 \equiv 1 \pmod{15}$ .

El tamaño de  $Z_{\text{mod } n}$  se denota  $\phi(n)$ . Esta función, conocida como función phi de Euler, satisface la ecuación

$$\phi(n) = \prod_{p \leq n} \frac{p-1}{p} \quad (31.20)$$

p W p es primo y p j n

de modo que p recorre todos los números primos que dividen a n (incluido el propio n, si n es primo). No probaremos aquí esta fórmula. Intuitivamente, empezamos con una lista de los n restos  $0, 1, 2, \dots, n-1$  y luego, por cada primo p que divide a n, tacha todos los múltiplos de p en la lista. Por ejemplo, dado que los divisores primos de 45 son 3 y 5,

$$\begin{array}{rccccc} .45 & / & D & 45 & 1 & \frac{1}{3} & 1 & \frac{1}{5} \\ & & & & & & & \\ D & 452 & & & & \frac{4}{3} & & \\ & & & & & & & \\ & & D & 24 & & & & \end{array}$$

Si p es primo, entonces  $Z_{\text{mod } p}$  re  $f_1; 2; \dots; p-1$ , y

$$\begin{array}{rccccc} .p & / & D & p & 1 & \frac{1}{p} & \\ & & & & & & \\ & & D & p & 1 & & \end{array} \quad (31.21)$$

Si n es compuesto, entonces  $\phi(n) < n-1$ , aunque se puede demostrar que

$$\phi(n) > \frac{n}{e \ln \ln n} \quad (31.22)$$

para  $n \geq 3$ , donde  $e \approx 2.718281828459045$  es la constante de Euler. Un límite inferior algo más simple (pero más flexible) para  $n > 5$  es

$$\phi(n) > \frac{n}{6 \ln \ln n} \quad (31.23)$$

Límite inferior (31.22) es esencialmente el mejor posible, ya que

$$\liminf_{n \rightarrow \infty} \frac{\phi(n)}{n \ln \ln n} = e^{-\gamma} \quad (31.24)$$

### subgrupos

Si  $S$ ;  $\circ$  es un grupo,  $S_0$  es un subgrupo, entonces  $S_0$ ;  $\circ$  es también un grupo, entones  $S_0$ ;  $\circ$  es un subgrupo de  $S$ ;  $\circ$ . Por ejemplo, los enteros pares forman un subgrupo de los enteros bajo la operación de suma. El siguiente teorema proporciona una herramienta útil para reconocer subgrupos.

**Teorema 31.14 (Un subconjunto cerrado no vacío de un grupo finito es un subgrupo)**

Si  $.S; \cdot$  es un grupo finito y  $S_0$  es cualquier subconjunto no vacío de  $S$  tal que  $a \cdot b \in S_0$  para todo  $a, b \in S_0$ , luego  $.S_0; \cdot$  es un subgrupo de  $.S; \cdot$ .

Demostración Dejamos la demostración como Ejercicio 31.3-3. ■

Por ejemplo, el conjunto  $\{0, 2, 4, 6\}$  forma un subgrupo de  $Z_8$ , ya que no está vacío y cerrado bajo la operación C (es decir, está cerrado bajo C8).

El siguiente teorema proporciona una restricción extremadamente útil sobre el tamaño de un subgrupo; omitimos la demostración.

**Teorema 31.15 (teorema de Lagrange)**

Si  $.S; \cdot$  es un grupo finito y  $.S_0; \cdot$  es un subgrupo de  $.S; \cdot$ , entonces  $|S_0|$  es un divisor de  $|S|$ . ■

Un subgrupo  $S_0$  de un grupo  $S$  es un subgrupo propio si  $S_0 \neq S$ . Usaremos el siguiente corolario en nuestro análisis en la Sección 31.8 del procedimiento de prueba de primalidad de Miller-Rabin.

**Corolario 31.16 Si**

$S_0$  es un subgrupo propio de un grupo finito  $S$ , entonces  $|S_0| = 2$ . ■

**Subgrupos generados por un elemento**

El teorema 31.14 nos da una manera fácil de producir un subgrupo de un grupo finito  $.S; \cdot$ : elige un elemento  $a$  y toma todos los elementos que se pueden generar a partir de  $a$  usando la operación de grupo. Específicamente, defina  $a^k$  para  $k \geq 1$  por

$$\begin{array}{c} k \\ \text{contacto directo/ DM} \\ a^k \\ \text{id1} \end{array} \quad \text{Gama} \quad \begin{array}{c} \text{un} \cdot \text{un} \cdot \text{un} \\ \vdots \\ \text{k} \end{array}$$

Por ejemplo, si tomamos un D 2 en el grupo Z6, la secuencia  $a.1/; a.2/; a.3/; \dots$  es

$2; 4; 0; 2; 4; 0; 2; 4; 0; \dots$

En el grupo  $Z_n$  tenemos  $a^k \equiv D_k \pmod{n}$ , y en el grupo  $Z$  tenemos  $a^k \equiv D_k \pmod{n}$ . Definimos  $\langle a \rangle$  como el subgrupo generado por  $a$ , denotado  $\langle a \rangle$  o  $\langle a \rangle; \cdot$ , por  $\langle a \rangle = \{a^k \mid k \in \mathbb{Z}\}$ . Decimos que  $a$  genera el subgrupo  $\langle a \rangle$ .

que  $a$  es generador de  $\langle a \rangle$ . Dado que  $S$  es finito,  $\langle a \rangle$  es un subconjunto finito de  $S$ , que posiblemente incluya todo  $S$ . Dado que la asociatividad de  $\cdot$  implica

$a_i / \circ a_j / D a_i C_j /$

haiis cerrado y por lo tanto, por el Teorema 31.14, haiis un subgrupo de S. Por ejemplo, en Z6, tenemos

$h_0i D f_0g ; h_1i$

$Df_0; 1; 2; 3; 4; 5g; h_2i Df_0; 2;$

$4g:$

De manera  $\therefore$ , tenemos

similar, en Z  $h_1i$

$D f_1g ; h_2i D f_1; 2; 4g;$

$h_3i D f_1; 2; 3; 4; 5; 6g:$

El orden de a (en el grupo S), denotado  $\text{ord.}a/$ , se define como el menor positivo entero t tal que en /  $\therefore$  D e.

### Teorema 31.17

Para cualquier grupo finito  $S; \circ$  y cualquier a  $\in S$ , el orden de a es igual al tamaño del subgrupo que genera, u  $\text{ord.}a/ D jhaij$ .

Prueba Let t D  $\text{ord.}a/$ . Como  $a^t / D e$  y  $a \cdot t C_k / D a^k /$  para  $k \geq 1$ , si  $i > t$ , entonces  $a^i / D a^j /$  para algún  $j < i$ . Así, como generamos elementos por a, no vemos elementos nuevos después de /. Así,  $haiDfa.1/; a.2/; \dots; at/g$ , y así  $jhaij t$ . Para demostrar que  $jhaij t$ , demostramos que cada elemento de la sucesión  $a.1/; a.2/; \dots; en/$  es distinto. Supongamos con el propósito de la contradicción que  $a^i / D a^j /$  para algunos i y j que satisfacen  $1 < i < j < t$ . Entonces,  $a \cdot i C_k / D a \cdot j C_k /$  para  $k \geq 0$ . Pero esta igualdad implica que  $a \cdot i C_k // D a \cdot j C_k // D e$ , una contradicción, ya que  $i \not\equiv j \pmod t$  pero t es el menor valor positivo tal que en / por delante, cada elemento de la secuencia  $a.1/; a.2/; \dots; at/$  es distinto, y  $jhaij$  concluyen que  $\text{ord.}a/ D jhaij$ .  $\therefore$  D e. Allí t.

Nosotros

■

### Corolario 31.18 La

secuencia  $a.1/; a.2/; \dots;$  es periódico con período t D  $\text{ord.}a/$ ; es decir,  $a^i / D a^j /$  si y solo si  $i \equiv j \pmod t$ .

■

De acuerdo con el corolario anterior, definimos  $a.0/$  como e y  $a^i /$  como  $a^i \pmod t$ , donde t D  $\text{ord.}a/$ , para todos los enteros i.

### Corolario 31.19

Si  $S; \circ$  es un grupo finito con identidad e, entonces para todo a  $\in S$ ,

$a \cdot j S_j / D e$

Demostración El teorema de Lagrange (Teorema 31.15) implica que  $\text{ord.}a/ j \mid \phi_j$ , y por lo tanto  $\phi_j \mid 0 \pmod t$ , donde  $t \leq \text{ord.}a$ . Por lo tanto,  $a \cdot \phi_j \mid a \cdot 0 \mid D$  e. ■

### Ejercicios

#### 31.3-1

Dibuje las tablas de operación de grupos para los grupos  $\mathbb{Z}_4$ ;  $\mathbb{C}_4$  y  $\mathbb{Z}_5$ . Muestre que estos grupos son isomorfos mostrando una correspondencia uno a uno entre sus elementos tal que a  $\mapsto b$   $\pmod 4$  si y solo si  $a \cdot b \equiv c \pmod 5$ .

#### 31.3-2

Enumere todos los subgrupos de  $\mathbb{Z}_9$  y de  $\mathbb{Z}_3$

#### 31.3-3

Demostrar el teorema 31.14.

#### 31.3-4

Muestre que si  $p$  es primo y  $e$  es un entero positivo, entonces

$$\text{pe} \mid D \text{ pe}^e \mid 1 :$$

#### 31.3-5

Muestre que para cualquier entero  $n > 1$  y para cualquier  $a \in \mathbb{Z}$  definido por  $f_a(x) = ax \pmod n$  es una permutación de  $\mathbb{Z}_{n-1}$ . La función  $f_a$  es biyectiva de  $\mathbb{Z}_{n-1}$  a  $\mathbb{Z}_{n-1}$ .

### 31.4 Resolución de ecuaciones lineales modulares

Consideremos ahora el problema de encontrar soluciones a la ecuación

$$ax \equiv b \pmod n ; \quad (31.25)$$

donde  $a > 0$  y  $n > 0$ . Este problema tiene varias aplicaciones; por ejemplo, lo usaremos como parte del procedimiento para encontrar claves en el criptosistema de clave pública RSA en la Sección 31.7.

Suponemos que se dan  $a$ ,  $b$  y  $n$ , y deseamos encontrar todos los valores de  $x$ , módulo  $n$ , que satisfagan la ecuación (31.25). La ecuación puede tener cero, uno o más de una de esas soluciones.

Sea  $H$  el subgrupo de  $\mathbb{Z}_n$  generado por  $a$ . Como  $H = \langle a \rangle$   $x \equiv b \pmod n$  si y sólo si  $a^x \equiv b \pmod n$ . El teorema de La Grange (Teorema 31.15) nos dice que  $a^x \equiv b \pmod n$  si y sólo si  $a^{\phi(n)} \equiv 1 \pmod n$  y  $b^{\phi(n)} \equiv a^{-1} \pmod n$ . La ecuación (31.25) tiene soluciones si y sólo si  $a^{\phi(n)} \equiv 1 \pmod n$  y  $b^{\phi(n)} \equiv a^{-1} \pmod n$ .

**Teorema 31.20**

Para cualquier número entero positivo  $a \neq 0$ , si  $d \mid \gcd(a; n)$ , entonces

$$\text{hai} \equiv \text{Df}_0; d; 2d; \dots; n=d / 1/dg \quad (31.26)$$

en  $Z_n$ , y por lo tanto

$$jhai \equiv r \pmod{n}$$

**Prueba** Comenzamos demostrando que  $d \mid \text{hai}$ . Recuérdese que EXTENDED-EUCLID( $a; n$ ) produce números enteros  $x_0$  e  $y_0$  tales que  $ax_0 + ny_0 \equiv 1 \pmod{d}$ . Así,  $ax_0 \equiv 1 \pmod{d}$ , de modo que  $d \mid \text{hai}$ . En otras palabras,  $d$  es múltiplo de  $a$  en  $Z_n$ .

Como  $d \mid \text{hai}$ , se sigue que todo múltiplo de  $d$  pertenece a  $\text{hai}$ , porque todo múltiplo de un múltiplo de  $a$  es en sí mismo un múltiplo de  $a$ . Así,  $\text{hai}$  contiene todos los elementos de  $f_0; d; 2d; \dots; n=d / 1/dg$ . Es decir,  $\text{hdihai}$ .

Ahora mostramos que  $\text{hdihai} = \text{hai}$ . Si  $m \equiv \text{hai} \pmod{d}$ , entonces  $m \equiv ax \pmod{n}$  para algún entero  $x$ , y por lo tanto  $m \equiv ax \pmod{d}$  para algún entero  $y$ . Sin embargo,  $d \mid a$  y  $d \mid n$ , y así  $d \mid m$  por la ecuación (31.4). Por lo tanto,  $m \equiv \text{hai} \pmod{d}$ .

Combinando estos resultados, tenemos que  $\text{hai} \equiv \text{hdihai}$ . Para ver que  $jhai \equiv r \pmod{n}$ , observe que hay exactamente  $n=d$  múltiplos de  $d$  entre  $0$  y  $n-1$ , inclusive. ■

**Corolario 31.21 La**

ecuación  $ax + b \equiv 0 \pmod{n}$  tiene solución para la incógnita  $x$  si y sólo si  $d \mid b$ , donde  $d = \gcd(a; n)$ .

**Demostración** La ecuación  $ax + b \equiv 0 \pmod{n}$  es solucionable si y sólo si  $b \equiv 0 \pmod{d}$ , que es lo mismo que decir

$$b \equiv 0 \pmod{d}$$

por el Teorema 31.20. Si  $0 < b < n$ , entonces  $b \not\equiv 0 \pmod{d}$  si y sólo si  $d \nmid b$ , ya que los miembros de  $\text{hai}$  son precisamente los múltiplos de  $d$ . Si  $b < 0$  o  $b \geq n$ , el corolario se sigue de la observación de que  $d \mid b$  si y sólo si  $b \equiv 0 \pmod{d}$ , ya que  $b \equiv b - n \pmod{d}$ .

**Corolario 31.22 La**

ecuación  $ax + b \equiv 0 \pmod{n}$  tiene  $d$  soluciones distintas módulo  $n$ , donde  $d = \gcd(a; n)$ , o no tiene soluciones.

**Prueba** Si  $ax + b \equiv 0 \pmod{n}$  tiene solución, entonces  $b \equiv 0 \pmod{d}$ . Por el Teorema 31.17,  $\text{ord}(a) \mid D_{jhai}$ , y así el Corolario 31.18 y el Teorema 31.20 implican que la secuencia es periódica con período aparece exactamente  $d$  veces en  $\text{hai} \pmod{d}$ . Si  $b \equiv 0 \pmod{d}$ , entonces  $b \equiv ai \pmod{n}$ , para  $i \in \{0, 1, \dots, d-1\}$ , en la secuencia  $ai \pmod{n}$ , para  $i \in \{0, 1, \dots, d-1\}$ , ya que

el bloque de valores  $\text{length-}n=d / \text{hai}$  se repite exactamente  $d$  veces a medida que  $i$  aumenta de 0 a  $n-1$ . Los índices  $x$  de las  $d$  posiciones para las cuales  $ax \bmod n \equiv b$  son las soluciones de la ecuación  $ax \equiv b \pmod{n}$ . ■

### Teorema 31.23

Sea  $d \mid \text{gcd}(a; n)$ , y supongamos que  $d \mid ax_0 \pmod{n}$  para algunos enteros  $x_0$  e  $y_0$  (por ejemplo, calculados por EXTENDED-EUCLID). Si  $d \mid b$ , entonces la ecuación  $ax \equiv b \pmod{n}$  tiene como una de sus soluciones el valor  $x_0$ , donde

$$x_0 \equiv x_0 \pmod{b=d \mid \text{mod } n}$$

Prueba que tenemos

$$\begin{aligned} \text{hacha0} & \quad ax_0 \equiv b \pmod{n} \\ db & \equiv d \pmod{n} \quad |b \pmod{n} \quad (\text{porque } ax_0 \equiv d \pmod{n}) \\ n & ; \end{aligned}$$

y por tanto  $x_0$  es una solución de  $ax \equiv b \pmod{n}$ . ■

### Teorema 31.24

Suponga que la ecuación  $ax \equiv b \pmod{n}$  tiene solución (es decir,  $d \mid b$ , donde  $d = \text{gcd}(a; n)$ ) y que  $x_0$  es cualquier solución de esta ecuación. Entonces, esta ecuación tiene exactamente  $d$  soluciones distintas, módulo  $n$ , dadas por  $x_i \equiv x_0 \pmod{d}$  para  $i = 0; 1; \dots; d-1$ .

Prueba Porque  $n=d > 0$  y  $0 \leq i \leq d-1$  para  $i = 0; 1; \dots; d-1$ , los valores  $x_0; x_1; \dots; x_{d-1}$  son todos distintos, módulo  $n$ . Como  $x_0$  es una solución de  $ax \equiv b \pmod{n}$ , tenemos  $ax_0 \equiv b \pmod{n}$ . Así, para  $i = 0; 1; \dots; d-1$ , tenemos

$$ax_i \equiv x_0 \pmod{d} \quad a|x_0 \pmod{d} \quad ax_0 \equiv b \pmod{n}$$

$$\begin{aligned} C & \quad a|x_0 \pmod{d} \quad ax_0 \equiv b \pmod{n} \\ n & \quad (\text{porque } d \mid a \text{ implica que } a|x_0 \text{ es un múltiplo de } n) \\ & \quad b \pmod{n} ; \end{aligned}$$

y por lo tanto  $ax_i \equiv b \pmod{n}$ , haciendo que  $x_i$  también sea una solución. Por el Corolario 31.22, la ecuación  $ax \equiv b \pmod{n}$  tiene exactamente  $d$  soluciones, de modo que  $x_0; x_1; \dots; x_{d-1}$  debe ser todos ellos. ■

Ahora hemos desarrollado las matemáticas necesarias para resolver la ecuación  $ax \equiv b \pmod{n}$ ; el siguiente algoritmo imprime todas las soluciones a esta ecuación. Las entradas  $a$  y  $n$  son números enteros positivos arbitrarios y  $b$  es un número entero arbitrario.

```

MODULAR-LINEAR-ECUATION-SOLVER.a; b; n/ 1 .d; x0 ;
y0 /D EXTENDIDO-EUCLID.a; n/ 2 if djb 3 x0 D
x0 .b=d /
mod n 4 for i D 0 to d 1 5 print .x0
C in=d // mod n 6 else imprime
"sin soluciones"

```

Como ejemplo de la operación de este procedimiento, considere la ecuación  $14x \equiv 30 \pmod{100}$  (aquí,  $a = 14$ ,  $b = 30$  y  $n = 100$ ). Llamando EUCLID EXTENDIDO en la línea 1, obtenemos  $d; x_0 ; y_0 / D .2; 7; 1/$ . Como  $2 \mid 30$ , se ejecutan las líneas 3–5. La línea 3 calcula  $x_0 D .7/15 \pmod{100} D 95$ . El bucle en las líneas 4 y 5 imprime las dos soluciones 95 y 45.

El procedimiento MODULAR-LINEAR-ECUATION-SOLVER funciona de la siguiente manera. La línea 1 calcula  $d = \text{mcd}(a; n)$ , junto con dos valores  $x_0$  e  $y_0$  tales que  $d \mid ax_0 + ny_0$ , demostrando que  $x_0$  es una solución a la ecuación  $ax \equiv b \pmod{n}$ . Si  $d \nmid a$ , entonces la ecuación  $ax \equiv b \pmod{n}$  no tiene solución, por el Corolario 31.21. La línea 2 comprueba si  $d \mid b$ ; si no, la línea 6 informa que no hay soluciones. De lo contrario, la línea 3 calcula una solución  $x_0$  para  $ax \equiv b \pmod{n}$ , de acuerdo con el Teorema 31.23. Dada una solución, el teorema 31.24 establece que sumando múltiplos de  $n/d$ , módulo  $n$ , se obtienen las otras soluciones. El ciclo for de las líneas 4 y 5 imprime todas las soluciones  $d$ , comenzando con  $x_0$  y con una separación  $n/d$ , módulo  $n$ .

MODULAR-LINEAR-ECUATION-SOLVER realiza  $O(\lg n)$  operaciones aritméticas, ya que EXTENDED-EUCLID realiza operaciones aritméticas  $O(\lg n)$ , y cada iteración del ciclo for de las líneas 4–5 realiza un número constante de operaciones aritméticas.

Los siguientes corolarios del Teorema 31.24 dan especializaciones de particular interés.

#### Corolario 31.25

Para cualquier  $n > 1$ , si  $\text{mcd}(a; n) \neq 1$ , entonces la ecuación  $ax \equiv b \pmod{n}$  tiene solución única, módulo  $n$ . ■

Si  $b \equiv 1 \pmod{n}$ , un caso común de considerable interés, la  $x$  que buscamos es un inverso multiplicativo de  $a$ , módulo  $n$ .

#### Corolario 31.26

Para cualquier  $n > 1$ , si  $\text{mcd}(a; n) \neq 1$ , entonces la ecuación  $ax \equiv 1 \pmod{n}$  tiene solución única, módulo  $n$ . De lo contrario, no tiene solución. ■

Gracias al Corolario 31.26, podemos usar la notación  $a^{-1} \bmod n$  para referirnos al inverso multiplicativo de  $a$ , módulo  $n$ , cuando  $a$  y  $n$  son primos relativos. Si  $\gcd(a; n) \neq 1$ , entonces la única solución a la ecuación  $ax \equiv 1 \pmod{n}$  es el entero  $x$  devuelto por EXTENDED-EUCLID, ya que la ecuación

$$\gcd(a; n) \neq 1 \text{ implica } C \neq ny$$

usando  $x \equiv a^{-1} \pmod{n}$ . Por lo tanto, podemos calcular  $a^{-1} \bmod n$  de manera eficiente implica  $ax \equiv 1 \pmod{n}$ .

### Ejercicios

#### 31.4-1

Encuentre todas las soluciones a la ecuación  $35x \equiv 10 \pmod{50}$ .

#### 31.4-2

Demuestre que la ecuación  $ax \equiv ay \pmod{n}$  implica  $xy \equiv 0 \pmod{n}$  siempre que  $\gcd(a; n) \neq 1$ . Muestre que la condición  $\gcd(a; n) \neq 1$  es necesario proporcionando un contraejemplo con  $\gcd(a; n) > 1$ .

#### 31.4-3

Considere el siguiente cambio a la línea 3 del procedimiento SOLUCIÓN DE ECUACIONES LINEALES -MODULARES:

$$3 \quad x_0 \equiv x_0 \pmod{b=d} \quad .n=d /$$

esto funcionara? Explica por qué o por qué no.

#### 31.4-4 ?

Sea  $p$  un primo y  $f(x) \in \mathbb{Z}_p[x]$  un polinomio de grado  $t$ , con coeficientes  $f_i$  extraídos de  $\mathbb{Z}_p$ . Decimos que  $a$  es un cero de  $f$  si  $f(a) \equiv 0 \pmod{p}$ . Demostrar que si  $a$  es un cero de  $f$ , entonces  $f(x) \equiv (x-a)^t \pmod{p}$  para algún polinomio  $g(x)$  de grado  $t-1$ . Demostrar por inducción sobre  $t$  que si  $p$  es primo, entonces un polinomio  $f(x)$  de grado  $t$  puede tener como máximo  $t$  ceros distintos módulo  $p$ .

## 31.5 El teorema chino del resto

Alrededor del año 100 dC, el matemático chino Sun-Tsú resolvió el problema de encontrar los números enteros  $x$  que dejan residuos 2, 3 y 2 cuando se dividen por 3, 5 y 7 respectivamente. Una de esas soluciones es  $x = 23$ ; todas las soluciones son de la forma  $23 + 105k$

para enteros arbitrarios  $k$ . El “teorema chino del resto” proporciona una correspondencia entre un sistema de ecuaciones módulo un conjunto de módulos primos relativamente primos (por ejemplo, 3, 5 y 7) y una ecuación módulo su producto (por ejemplo, 105).

El teorema chino del resto tiene dos aplicaciones principales. Sea el entero  $n$  factorizado como  $n \equiv n_1 n_2 \cdots n_k$ , donde los factores  $n_i$  son pares relativamente primos. Primero, el teorema chino del resto es un “teorema de estructura” descriptivo que describe la estructura de  $Z_n$  como idéntica a la del producto cartesiano  $Z_{n_1} Z_{n_2} \cdots Z_{n_k}$  con módulo  $n$  de suma y multiplicación por componentes en el  $i$ -ésimo componente. En segundo lugar, esta descripción nos ayuda a diseñar algoritmos eficientes, ya que trabajar en cada uno de los sistemas  $Z_{n_i}$  puede ser más eficiente (en términos de operaciones de bits) que trabajar en módulo  $n$ .

#### Teorema 31.27 (teorema del resto chino)

Sea  $n \equiv n_1 n_2 \cdots n_k$ , donde los  $n_i$  son pares relativamente primos. Considera la correspondencia

$$a \equiv a_1; a_2; \dots; a_k ; \quad (31.27)$$

donde  $a \equiv 2 \pmod{n_i}$ , y

$a_i \equiv d \pmod{n_i}$

para  $i \in \{1, 2, \dots, k\}$ . Entonces, el mapeo (31.27) es una correspondencia uno a uno (bijección) entre  $Z_n$  y el producto cartesiano  $Z_{n_1} Z_{n_2} \cdots Z_{n_k}$ . Las operaciones realizadas en los elementos de  $Z_n$  se pueden realizar de manera equivalente en las  $k$ -tuplas correspondientes realizando las operaciones de forma independiente en cada posición de coordenadas en el sistema apropiado. Es decir, si

$$a \equiv a_1; a_2; \dots; a_k ; \quad b \equiv b_1;$$

$$b_2; \dots; b_k ;$$

entonces

$$a + b \equiv a_1 + b_1 \pmod{n_1}; \dots; a_k + b_k \pmod{n_k} ; \quad (31.28)$$

$$a - b \equiv a_1 - b_1 \pmod{n_1}; \dots; a_k - b_k \pmod{n_k} ; \quad (31.29)$$

$$a \cdot b \equiv a_1 \cdot b_1 \pmod{n_1}; \dots; a_k \cdot b_k \pmod{n_k} ; \quad (31.30)$$

Prueba La transformación entre las dos representaciones es bastante sencilla.

Pasando de  $a \equiv a_1; a_2; \dots; a_k$  es bastante fácil y solo requiere  $k$  operaciones “mod”.

Calcular  $a$  a partir de las entradas  $a_1; a_2; \dots; a_k$  es un poco más complicado. Comenzamos definiendo  $m_i \equiv n_i^{-1}$  para  $i \in \{1, 2, \dots, k\}$ ; por lo tanto,  $m_i$  es el producto de todos los  $n_j$  que no sean  $n_i$ :  $m_i \equiv n_1 n_2 \cdots n_i \cdots n_k$ . A continuación definimos

$$ci \in D \text{ mi.m}_1 \bmod n_i \text{ para } i \quad (31.31)$$

$D = \{1; 2; \dots; k\}$ . La ecuación (31.31) siempre está bien definida: dado que  $m_i$  y  $n_i$  son primos relativos (por el teorema 31.6), el corolario 31.26 garantiza que  $m_1 \bmod n_i$  existe. Finalmente, podemos calcular  $a$  como una función de  $a_1, a_2, \dots, a_k$  como sigue:  $a = a_1c_1 + a_2c_2 + \dots + a_kc_k$

$\text{CC } ak \equiv \sum a_i c_i \pmod{n_i} : (31.32)$   $a_i \bmod n_i$  para  $i \in D$  Ahora mostramos que la ecuación (31.32)

que si  $j \neq i$ , entonces  $m_j \not\equiv 0 \pmod{n_i}$ , lo que implica que  $c_j = 0$ ; asegura que  $a \in K$ . Nótese Tenemos  $\sum a_i c_i \equiv 0 \pmod{n_i}$ . Nótese también que  $c_i \equiv 1 \pmod{n_i}$ , de la ecuación (31.31).

así la atractiva y útil correspondencia  $c_i = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$  es una función de  $a$ . Un vector que tiene ceros en todas partes excepto en la  $i$ -ésima coordenada, donde

tiene un 1; los  $c_i$  forman así una

“base” para la representación, en cierto sentido. Para cada  $i$ , por lo tanto, tenemos

$$\begin{aligned} a &= a_1c_1 + a_2c_2 + \dots + a_kc_k \pmod{n_i} \\ &\equiv a_1m_1 \pmod{n_i} + a_2m_2 \pmod{n_i} + \dots + a_km_k \pmod{n_i} ; \\ &= a_i \end{aligned}$$

que es lo que deseábamos mostrar: nuestro método de calcular  $a$  a partir de los  $a_i$  produce un resultado  $a$  que satisface las restricciones  $a \equiv a_i \pmod{n_i}$  para  $i \in D = \{1; 2; \dots; k\}$ .

La correspondencia es uno a uno, ya que podemos transformar en ambas direcciones.

Finalmente, las ecuaciones (31.28)–(31.30) se derivan directamente del ejercicio 31.1-7, ya que  $x \equiv a \pmod{n_i}$  para cualquier  $x \in D = \{1; 2; \dots; k\}$ . ■

Usaremos los siguientes corolarios más adelante en este capítulo.

#### Corolario 31.28 Si

$n_1, n_2, \dots, n_k$  son pares relativamente primos y  $n = n_1n_2 \dots n_k$ , entonces para cualquier número entero  $a_1, a_2, \dots, a_k$ , el conjunto de ecuaciones simultáneas  $x$

$$a_i \equiv a \pmod{n_i} ; \text{ para } i \in D$$

tiene una única solución módulo  $n$  para la incógnita  $x$ . ■

#### Corolario 31.29 Si

$n_1, n_2, \dots, n_k$  son pares relativamente primos y  $n = n_1n_2 \dots n_k$ , entonces para todos los enteros  $x$  y  $a$ ,  $x \equiv a \pmod{n}$

$$\text{para } i \in D = \{1; 2; \dots; k\}$$

$$\text{si y solo si } x \equiv a \pmod{n} :$$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0 0	40	15	55	30	5	45	20	60	35	10	50	25	1
51	2	52	27	2	42	17	57	32	7	47	22	62	37
63	38												
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Figura 31.3 Una ilustración del teorema chino del resto para  $n_1 \equiv 5 \pmod{D}$  y  $n_2 \equiv 13 \pmod{D}$ . Para este ejemplo,  $c_1 \equiv 26 \pmod{D}$  y  $c_2 \equiv 40 \pmod{D}$ . En la fila  $i$ , la columna  $j$  se muestra el valor de  $a_i \pmod{65}$ , tal que  $a_i \equiv j \pmod{5}$  y  $a_i \equiv c_j \pmod{13}$ . Tenga en cuenta que la fila 0, columna 0 contiene un 0. De manera similar, la fila 4, columna 12 contiene un 64 (equivalente a 1). Dado que  $c_1 \equiv 26 \pmod{D}$ , bajar una fila aumenta  $a_i$  en 26. De manera similar,  $c_2 \equiv 40 \pmod{D}$  significa que moverse una columna a la derecha aumenta  $a_i$  en 40. Aumentar  $a_i$  en 1 corresponde a moverse en diagonal hacia abajo y hacia la derecha, dando vueltas de abajo hacia arriba y de derecha a izquierda.

Como ejemplo de la aplicación del teorema chino del resto, supongamos que se dan las dos ecuaciones

$$a \equiv 2 \pmod{5};$$

$$a \equiv 3 \pmod{13};$$

de modo que  $a_1 \equiv 2 \pmod{2}$ ,  $n_1 \equiv 5 \pmod{D}$ ,  $a_2 \equiv 3 \pmod{13}$  y  $n_2 \equiv 13 \pmod{D}$ , y deseamos calcular  $a \pmod{65}$ , ya que  $n \equiv n_1 n_2 \pmod{65}$ . Porque  $131 \equiv 2 \pmod{5}$  y  $51 \equiv 3 \pmod{13}$ , tenemos

$$c_1 \equiv 13 \cdot 2 \pmod{5} \quad ;$$

$$c_2 \equiv 5 \cdot 3 \pmod{13} \quad ;$$

y

$$a \equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65} \quad .mod$$

$$52 \cdot 120 \pmod{65} \quad .mod$$

$$42 \pmod{65} \quad .mod$$

Consulte la figura 31.3 para ver una ilustración del teorema del resto chino, módulo 65.

Por lo tanto, podemos trabajar módulo  $n$  trabajando módulo  $n$  directamente o trabajando en la representación transformada usando cálculos de módulo ni separados, según convenga. Los cálculos son totalmente equivalentes.

### Ejercicios

#### 31.5-1

Encuentre todas las soluciones de las ecuaciones  $x \equiv 4 \pmod{5}$  y  $x \equiv 5 \pmod{11}$ .

## 31.5-2

Encuentre todos los enteros  $x$  que dejen restos 1, 2, 3 cuando se dividen por 9, 8, 7 respectivamente.

## 31.5-3

Argumente que, según las definiciones del teorema 31.27, si  $\text{gcd}(a; n) \neq 1$ ,  
 luego  $a_1 \text{ mod } n \mid a_1 \text{ mod } n_1; a_1 \text{ módulo } n_2; \dots; a_1 \text{ mod } n_k$ :

## 31.5-4

De acuerdo con las definiciones del teorema 31.27, demuestre que para cualquier polinomio  $f$ , el número de raíces de la ecuación  $f(x) \equiv 0 \pmod{n}$  es igual al producto del número de raíces de cada una de las ecuaciones  $f(x) \equiv 0 \pmod{n_1}$ ,  $f(x) \equiv 0 \pmod{n_2}$ , ...,  $f(x) \equiv 0 \pmod{n_k}$ .

## 31.6 Potencias de un elemento

Así como a menudo consideramos los múltiplos de un elemento dado  $a$ , módulo  $n$ , consideramos la secuencia de potencias de  $a$ , módulo  $n$ ,

donde  $a^i \pmod{n} : a^0; a^1; a^2; a^3; \dots$ ; (31.33)

módulo  $n$ . Indexando desde 0, el valor 0 en esta secuencia es  $a^0 \pmod{n} \neq 1$ , y el valor  $i$  es  $a^i \pmod{n}$ . Por ejemplo, las potencias de 3 módulo 7 son

$$\begin{array}{r} i \\ \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 3^i \pmod{7} & 1 & 3 & 2 & 6 & 4 & 5 & 1 & 3 & 2 & 6 & 4 & 5 \end{array}$$

mientras que las potencias de 2 módulo 7 son

$$\begin{array}{r} i \\ \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2^i \pmod{7} & 1 & 2 & 4 & 1 & 2 & 4 & 1 & 2 & 4 & 1 & 2 & 4 \end{array}$$

En esta sección, sea  $\langle a \rangle$  el subgrupo de  $\mathbb{Z}_n^\times$  generado por  $a$  por multiplicación repetida, y sea  $\text{ord}(a)$  (el “orden de  $a$ , módulo  $n$ ”) denote el orden en  $\mathbb{Z}_n^\times$ . Por ejemplo,  $h_{2^i} = 1$ ;  $2^4 \pmod{7} = 1$ ;  $4^2 \pmod{7} = 1$ . Usando la definición de la función  $\phi(n)$  como el tamaño de (ver Sección 31.3), ahora de Z Corolario 31.19 en la notación de traducimos obtener el teorema de Euler y especializarlo  $\mathbb{Z} \rightarrow \mathbb{Z}_p$ , donde  $p$  es primo, para obtener el teorema de Fermat.

Teorema 31.30 (teorema de Euler)

Para cualquier entero

$n > 1$ ,  $a^{n-1} \pmod{n}$  para todo  $a \in \mathbb{Z}_n^\times$



**Teorema 31.31 (teorema de Fermat)**

Si  $p$  es primo, entonces

$a^{p-1} \equiv 1 \pmod{p}$  para todo  $a \in \mathbb{Z}$

Demostración Por la ecuación (31.21),  $a^p \equiv 1 \pmod{p}$  si  $p$  es primo. ■

El teorema de Fermat se aplica a todos los elementos en  $\mathbb{Z}_p$  excepto 0, ya que  $0^{p-1} \not\equiv 1 \pmod{p}$ . Para todos los enteros  $a \neq 0$ , tenemos  $a^{p-1} \equiv 1 \pmod{p}$ .

Si  $a$  es una raíz primitiva de  $\mathbb{Z}_p$ , entonces cada elemento en  $\mathbb{Z}_p$  es una potencia de  $a$ , módulo  $p$ . Por ejemplo, 3 es una raíz primitiva de  $\mathbb{Z}_{13}$ , ya que  $3^1 \equiv 3 \pmod{13}$ ,  $3^2 \equiv 9 \pmod{13}$ ,  $3^3 \equiv 1 \pmod{13}$ , pero 2 no es una raíz primitiva de  $\mathbb{Z}_{13}$ . Si  $\mathbb{Z}_p$  posee una raíz primitiva, el grupo  $\mathbb{Z}_p^\times$  es cíclico. Omitimos la prueba del siguiente teorema, que es probado por Niven y Zuckerman [265]. ■

**Teorema 31.32** Los

valores de  $n > 1$  para los cuales  $\mathbb{Z}_n$  es cíclico son 2, 4,  $p - 1$  y  $2p - 1$ , para todos los números primos  $p$ .

Si  $g$  es una raíz primitiva de  $\mathbb{Z}_n$  y  $a$  es cualquier elemento de  $\mathbb{Z}_n$ , entonces existe un  $\ell$  tal que  $g^\ell \equiv a \pmod{n}$ . Este  $\ell$  es un logaritmo discreto o un índice de  $a$ , módulo  $n$ , en base  $g$ ; denotamos este valor como  $\text{ind}_n(ga)$ .

**Teorema 31.33 (Teorema del logaritmo discreto)**

Si  $g$  es una raíz primitiva de  $\mathbb{Z}_n$  sólo si  $n = p^k$ , entonces la ecuación  $gx \equiv gy \pmod{n}$  se cumple si y solo si  $x \equiv y \pmod{n}$ .

Prueba Suponga primero que  $x \not\equiv y \pmod{n}$ . Entonces,  $x \equiv Ckn \pmod{n}$  para algún entero  $k$ . Por lo tanto,  $gy \equiv Ckn \pmod{n}$ .

$$\begin{aligned} gx &\equiv gy \pmod{n} \\ &\equiv g(Ckn) \pmod{n} \\ &\equiv gkn \pmod{n} \\ &\equiv kn \pmod{n} \\ &\equiv 0 \pmod{n} \end{aligned}$$

Por el contrario, suponga que  $x \not\equiv y \pmod{n}$ . Debido a que la secuencia de potencias de  $g$  genera cada elemento de  $\mathbb{Z}_n$ , el Corolario 31.18 implica que la secuencia de potencias de  $g$  es periódica con período  $n$ . Por lo tanto, si  $gx \equiv gy \pmod{n}$ , entonces debemos tener  $x \equiv y \pmod{n}$ . ■

Ahora dirigimos nuestra atención a las raíces cuadradas de 1, módulo a potencia prima. El siguiente teorema será útil en nuestro desarrollo de un algoritmo de prueba de primalidad en la Sección 31.8.

Teorema 31.34 Si

$p$  es un primo impar y  $e \geq 1$ , entonces la ecuación  $x^2 \equiv 1 \pmod{p}$

$\equiv p-1$

(31.34)

tiene sólo dos soluciones, a saber,  $x \equiv 1$  y  $x \equiv -1$ .

La ecuación de prueba (31.34) es equivalente a

$\text{gcd}(p-1, x-1) = 1$ :

Como  $p > 2$ , podemos tener  $\text{gcd}(p-1, x-1) = 1$  o  $\text{gcd}(p-1, x-1) = p-1$ , pero no ambos. (De lo contrario, por la propiedad (31.3),  $p$  también dividiría su diferencia  $x-1$ .)

Si  $p-1 = d$ , entonces  $\text{gcd}(p-1, x-1) = d$ . Por el Corolario 31.5, tendríamos  $\text{gcd}(p-1, x-1) = 1$ . Es decir,  $x-1 \equiv 0 \pmod{d}$ . Simétricamente, si  $p-1 = d$ , entonces  $\text{gcd}(p-1, x-1) = d$ . Por el Corolario 31.5 implica que  $\text{gcd}(p-1, x-1) = 1$ , de modo que  $x-1 \equiv 0 \pmod{d}$ . Por lo tanto,  $x-1 \equiv 0 \pmod{p-1}$ .

■

Un número  $x$  es una raíz cuadrada no trivial de 1, módulo  $n$ , si satisface la ecuación  $x^2 \equiv 1 \pmod{n}$  pero  $x \not\equiv 1 \pmod{n}$  y  $x \not\equiv -1 \pmod{n}$ . Por ejemplo, 6 es una raíz cuadrada no trivial de 1, módulo 35.

Usaremos el siguiente corolario del teorema 31.34 en la prueba de corrección de la sección 31.8 para el procedimiento de prueba de primalidad de Miller-Rabin.

Corolario 31.35 Si

existe una raíz cuadrada no trivial de 1, módulo  $n$ , entonces  $n$  es compuesto.

Demostración Por la contrapositiva del Teorema 31.34, si existe una raíz cuadrada no trivial de 1, módulo  $n$ , entonces  $n$  no puede ser un primo impar o una potencia de un primo impar.

Si  $x^2 \equiv 1 \pmod{n}$ , entonces  $x \equiv 1 \pmod{n}$  o  $x \equiv -1 \pmod{n}$ , por lo que todas las raíces cuadradas de 1, módulo  $n$ , son triviales. Por lo tanto,  $n$  no puede ser primo. Finalmente, debemos tener  $n > 1$  para que exista una raíz cuadrada no trivial de 1. Por lo tanto,  $n$  debe ser compuesto.

■

Elevando a potencias con elevaciones repetidas al cuadrado

Una operación que ocurre con frecuencia en los cálculos teóricos de números es elevar un número a un módulo de potencia de otro número, también conocido como exponentiación modular. Más precisamente, nos gustaría una forma eficiente de calcular  $a^b \pmod{n}$ , donde  $a$  y  $b$  son números enteros no negativos y  $n$  es un número entero positivo. La exponentiación modular es una operación esencial en muchas rutinas de prueba de primalidad y en el criptosistema de clave pública RSA. El método de elevar al cuadrado repetidamente resuelve este problema de manera eficiente usando la representación binaria de  $b$ .

Sea  $b = b_k b_{k-1} \dots b_1 b_0$  sea la representación binaria de  $b$ . (Es decir, la representación binaria es  $k$  bits de largo,  $b_k$  es el bit más significativo y  $b_0$  es el menos

yo 98 7	6	5	4	3	2	1	0
bi 10 0 c 1 2 4 17	0	1	1				
d 7 49 157 526 160 241 298 166		35		0	70	140	280 560
67						1	

Figura 31.4 Los resultados de EXPONENCIACIÓN MODULAR al calcular  $a^b \mod n$ , donde  $a \equiv 7 \pmod{D}$ ,  $b \equiv 560 \pmod{D}$  y  $n \equiv 561 \pmod{D}$ . Los valores se muestran después de cada ejecución del bucle for. El resultado final es 1.

bit significativo.) El siguiente procedimiento calcula  $a^b \mod n$  a medida que  $c$  aumenta por duplicaciones e incrementos de 0 a  $b$ .

#### EXPONENCIACIÓN MODULAR.a; b; norte/

1 c D 0 2 d

D 1 3 let

hbk; bk1;:::;b0i sea la representación binaria de  $b$  para  $i \in D$  k hasta 0 5 c D 2c 6 d D .dd /

mod n if bi == 1

7 c D c C 1 d D .da/ mod n 10

retorno d

8

9

El uso esencial de elevar al cuadrado en la línea 6 de cada iteración explica el nombre de "cuadrado repetido". Como ejemplo, para  $a \equiv 7 \pmod{D}$ ,  $b \equiv 560 \pmod{D}$  y  $n \equiv 561 \pmod{D}$ , el algoritmo calcula la secuencia de valores módulo 561 que se muestra en la figura 31.4; la secuencia de exponentes utilizados aparece en la fila de la tabla etiquetada por  $c$ .

La variable  $c$  no es realmente necesaria para el algoritmo, pero se incluye para el siguiente ciclo invariante de dos partes:

Justo antes de cada iteración del bucle for de las líneas 4 a 9,

1. El valor de  $c$  es el mismo que el prefijo hbk; bk1;:::;biC1i de la representación binaria de  $b$ , y 2. d D ac mod n.

Usamos este ciclo invariante de la siguiente manera:

Inicialización: Inicialmente,  $i \in D$  k, por lo que el prefijo hbk; bk1;:::; biC1i está vacío, que corresponde a  $c \equiv 0 \pmod{D}$ . Además,  $d \equiv 1 \pmod{D}$   $a^0 \equiv 1 \pmod{n}$ .

Mantenimiento: Deje que  $c_0$  y  $d_0$  denotan los valores de  $c$  y  $d$  al final de una iteración del ciclo for y, por lo tanto, los valores antes de la siguiente iteración. Cada iteración actualiza  $c \leftarrow c \cdot D + 2c$  (si  $b_i = 0$ ) o  $c \leftarrow c \cdot D + 2c + 1$  (si  $b_i = 1$ ), de modo que  $c$  será correcto mod  $n$ .  $d \leftarrow d \cdot ac/2 + d \cdot 2a$  mod  $n$  antes de la siguiente iteración. Si  $b_i = 1$ , entonces  $d \leftarrow d \cdot ac/2 + d \cdot 2a + 1$  mod  $n$ . Si  $b_i = 0$ , entonces  $d \leftarrow d \cdot ac/2$  mod  $n$ . En cualquier caso,  $d \equiv ac \pmod{n}$  antes de la siguiente iteración.

Terminación: En la terminación,  $i = D - 1$ . Así,  $c \equiv D \cdot b$ , ya que  $c$  tiene el valor del prefijo  $hbk; bk_1;\dots;b_0$  de la representación binaria de  $b$ . Por lo tanto,  $d \equiv ac \pmod{n}$ .

Si las entradas  $a$ ,  $b$  y  $n$  son números de  $\lceil \log n \rceil$  bits, entonces el número total de operaciones aritméticas requeridas es  $O(\lceil \log n \rceil^2)$  y el número total de operaciones de bit requeridas es  $O(\lceil \log n \rceil^3)$ .

### Ejercicios

#### 31.6-1

Dibuje una tabla que muestre el orden de cada elemento en  $Z_{11}$ . Elija el primitivo más pequeño raíz  $g$  y calcule una tabla que dé  $ind_{11}(gx)$  para todo  $x \in Z_{11}$

#### 31.6-2

Proporcione un algoritmo de exponentiación modular que examine los bits de  $b$  de derecha a izquierda en lugar de izquierda a derecha.

#### 31.6-3

Suponiendo que sabe  $\phi(n)$ , explique cómo calcular  $a^e \pmod{n}$  para cualquier  $a \in Z$  usando el procedimiento MODULAR-EXPONENTIATION.

## 31.7 El criptosistema de clave pública RSA

Con un criptosistema de clave pública, podemos cifrar los mensajes enviados entre dos partes que se comunican para que un intruso que escuche los mensajes cifrados no pueda decodificarlos. Un criptosistema de clave pública también permite a una parte agregar una "firma digital" infalsificable al final de un mensaje electrónico.

Tal firma es la versión electrónica de una firma manuscrita en un documento en papel. Puede ser fácilmente verificado por cualquiera, falsificado por nadie, pero pierde su validez si se altera cualquier parte del mensaje. Por lo tanto, proporciona autenticación tanto de la identidad del firmante como del contenido del mensaje firmado. es la herramienta perfecta

para contratos comerciales firmados electrónicamente, cheques electrónicos, órdenes de compra electrónicas y otras comunicaciones electrónicas que las partes deseen autenticar.

El criptosistema de clave pública RSA se basa en la gran diferencia entre la facilidad de encontrar números primos grandes y la dificultad de factorizar el producto de dos números primos grandes. La Sección 31.8 describe un procedimiento eficiente para encontrar números primos grandes y la Sección 31.9 analiza el problema de factorizar números enteros grandes.

### Criptosistemas de clave pública

En un criptosistema de clave pública, cada participante tiene tanto una clave pública como una clave secreta. Cada clave es una pieza de información. Por ejemplo, en el criptosistema RSA, cada clave consta de un par de números enteros. Los participantes "Alice" y "Bob" se usan tradicionalmente en ejemplos de criptografía; denotamos sus claves públicas y secretas como PA, SA para Alice y PB , SB para Bob.

Cada participante crea sus propias claves públicas y secretas. Las claves secretas se mantienen en secreto, pero las claves públicas pueden revelarse a cualquier persona o incluso publicarse. De hecho, a menudo es conveniente suponer que la clave pública de todos está disponible en un directorio público, de modo que cualquier participante pueda obtener fácilmente la clave pública de cualquier otro participante.

Las claves pública y secreta especifican funciones que se pueden aplicar a cualquier mensaje. Sea D el conjunto de mensajes permisibles. Por ejemplo, D podría ser el conjunto de todas las secuencias de bits de longitud finita. En la formulación más simple y original de la criptografía de clave pública, requerimos que las claves pública y secreta especifiquen funciones uno a uno de D a sí mismo. Denotamos la función correspondiente a la clave pública PA de Alice por PA./ y la función correspondiente a su clave secreta SA por SA./. Las funciones PA./ y SA./ son entonces permutaciones de D. Suponemos que las funciones PA./ y SA./ son eficientemente computables dada la clave correspondiente PA o SA.

Las claves públicas y secretas de cualquier participante son un "par coincidente" en el sentido de que especifican funciones que son inversas entre sí. Eso es,

$$\text{MD } \text{SA.PA.M} // ; \text{ MD} \quad (31.35)$$

$$\text{PA.SA.M} // \quad (31.36)$$

para cualquier mensaje M 2 D. Transformando M con las dos claves PA y SA sucesivamente, en cualquier orden, se obtiene de nuevo el mensaje M.

En un criptosistema de clave pública, requerimos que nadie más que Alice pueda calcular la función SA./ en cualquier período de tiempo práctico. Esta suposición es crucial para mantener privado el correo cifrado enviado a Alice y para saber que las firmas digitales de Alice son auténticas. Alice debe mantener SA en secreto; si no lo hace, pierde su singularidad y el criptosistema no puede proporcionarle capacidades únicas.

La suposición de que solo Alice puede calcular SA./ debe mantenerse aunque todos

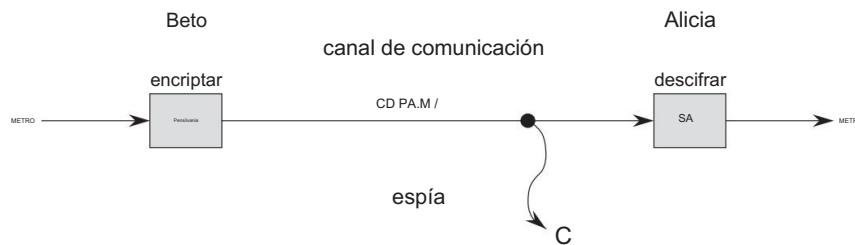


Figura 31.5 Cifrado en un sistema de clave pública. Bob encripta el mensaje M utilizando la clave pública PA de Alice y transmite el CD PA.M / de texto cifrado resultante a través de un canal de comunicación a Alice. Un intruso que captura el texto cifrado transmitido no obtiene información sobre M. Alice recibe C y lo descifra usando su clave secreta para obtener el mensaje original MD SA.C /.

conoce PA y puede calcular PA./, la función inversa de SA./, de manera eficiente. Para diseñar un criptosistema de clave pública viable, debemos descubrir cómo crear un sistema en el que podamos revelar una transformación PA./ sin revelar cómo calcular la transformación inversa correspondiente SA./. Esta tarea parece formidable, pero veremos cómo llevarla a cabo.

En un criptosistema de clave pública, el cifrado funciona como se muestra en la Figura 31.5. Supongamos que Bob desea enviar a Alice un mensaje M encriptado para que parezca un galimatías ininteligible a un intruso. El escenario para enviar el mensaje es el siguiente.

Bob obtiene la PA de clave pública de Alice (de un directorio público o directamente de Alice).

Bob calcula el CD de texto cifrado PA.M/ correspondiente al mensaje M y envía C a Alice.

Cuando Alice recibe el texto cifrado C, aplica su clave secreta SA para recuperar el mensaje original: SA.C / D SA.PA.M // D M.

Debido a que SA./ y PA./ son funciones inversas, Alice puede calcular M a partir de C. Debido a que solo Alice puede calcular SA./, Alice es la única que puede calcular M a partir de C. Debido a que Bob encripta M usando PA./, solo Alice puede entender el mensaje transmitido.

Podemos implementar firmas digitales con la misma facilidad dentro de nuestra formulación de un criptosistema de clave pública. (Hay otras formas de abordar el problema de la construcción de firmas digitales, pero no las abordaremos aquí.) Supongamos ahora que Alice desea enviar a Bob una respuesta M0 firmada digitalmente. La Figura 31.6 muestra cómo procede el escenario de la firma digital.

Alice calcula su firma digital para el mensaje M0 usando su clave secreta SA y la ecuación D SA.M0 /.

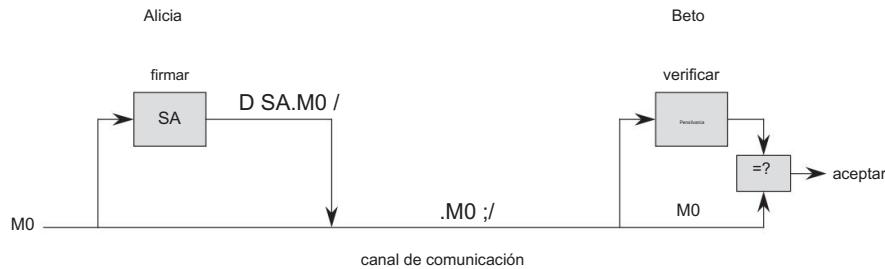


Figura 31.6 Firmas digitales en un sistema de clave pública. Alice firma el mensaje  $M_0$  añadiendo su firma digital  $D_{SA}M_0 /$  al mismo. Transmite el par mensaje/firma  $.M_0 ;/$  a Bob, quien lo verifica comprobando la ecuación  $M_0 D_{PA} /$ . Si la ecuación se cumple, acepta  $.M_0 ;/$  como un mensaje que Alice ha firmado.

Alice envía el par mensaje/firma  $.M_0 ;/$  a Bob.

Cuando Bob recibe  $.M_0 ;/$ , puede verificar que se originó en Alice usando la clave pública de Alice para verificar la ecuación  $M_0 D_{PA} /$ . (Presumiblemente,  $M_0$  contiene el nombre de Alice, por lo que Bob sabe qué clave pública usar). Si la ecuación se cumple, entonces Bob concluye que el mensaje  $M_0$  en realidad fue firmado por Alice. Si la ecuación no se cumple, Bob concluye que el mensaje  $M_0$  o la firma digital se corrompieron debido a errores de transmisión o que el par  $.M_0 ;/$  es un intento de falsificación.

Debido a que una firma digital proporciona tanto la autenticación de la identidad del firmante como la autenticación del contenido del mensaje firmado, es análoga a una firma manuscrita al final de un documento escrito.

Una firma digital debe ser verificable por cualquier persona que tenga acceso a la clave pública del firmante. Un mensaje firmado puede ser verificado por una parte y luego pasado a otras partes que también pueden verificar la firma. Por ejemplo, el mensaje podría ser un cheque electrónico de Alice a Bob. Después de que Bob verifique la firma de Alice en el cheque, puede entregar el cheque a su banco, quien también puede verificar la firma y efectuar la transferencia de fondos adecuada.

Un mensaje firmado no está necesariamente encriptado; el mensaje puede ser "claro" y no estar protegido contra la divulgación. Al componer los protocolos anteriores para el cifrado y las firmas, podemos crear mensajes que estén firmados y cifrados.

El firmante primero agrega su firma digital al mensaje y luego encripta el par mensaje/firma resultante con la clave pública del destinatario previsto. El destinatario descifra el mensaje recibido con su clave secreta para obtener tanto el mensaje original como su firma digital. El destinatario puede entonces verificar la firma utilizando la clave pública del firmante. El correspondiente proceso combinado utilizando sistemas basados en papel sería firmar el documento en papel y

Luego selle el documento dentro de un sobre de papel que solo abre el destinatario previsto.

### El criptosistema RSA

En el criptosistema de clave pública RSA, un participante crea sus claves pública y secreta con el siguiente procedimiento:

1. Seleccionar al azar dos números primos grandes  $p$  y  $q$  tales que  $p \neq q$ . Los números primos  $p$  y  $q$  pueden tener, digamos, 1024 bits cada uno.
2. Calcule  $n = p \cdot q$ .
3. Seleccione un pequeño entero impar  $e$  que sea primo relativo a  $n$ , el cual, por la ecuación (31.20), es igual a  $\phi(n)$ .
4. Calcule  $d$  como el inverso multiplicativo de  $e$ , módulo  $n$ . (El corolario 31.26 garantiza que  $d$  existe y está definida de manera única. Podemos usar la técnica de la sección 31.4 para calcular  $d$ , dados  $e$  y  $n$ ).
5. Publicar el par  $(e, n)$  como clave pública RSA del participante.
6. Mantener en secreto el par  $(d, n)$  como clave secreta RSA del participante.

Para este esquema, el dominio  $D$  es el conjunto  $Z_n$ . Transformar un mensaje  $M$  asociado a una clave pública  $(e, n)$ , calcular

$$C = M^e \pmod{n} \quad (31.37)$$

Transformar un texto cifrado  $C$  asociado a una clave secreta  $(d, n)$ , calcular

$$M = C^d \pmod{n} \quad (31.38)$$

Estas ecuaciones se aplican tanto al cifrado como a las firmas. Para crear una firma, el firmante aplica su clave secreta al mensaje a firmar, en lugar de a un texto cifrado. Para verificar una firma, se le aplica la clave pública del firmante, en lugar de un mensaje a cifrar.

Podemos implementar las operaciones de clave pública y clave secreta usando el procedimiento EXPONENCIACIÓN MODULAR descrito en la Sección 31.6. Para analizar el tiempo de ejecución de estas operaciones, suponga que la clave pública  $(e, n)$  y clave secreta  $(d, n)$  satisfacen  $\lg e = D$ ,  $\lg d = O(1)$  y  $\lg n = O(1)$ . Luego, la aplicación de una clave pública requiere  $O(D)$  multiplicaciones modulares y utiliza  $O(\lg n)$  operaciones de bits. La aplicación de una clave secreta requiere  $O(D)$  multiplicaciones modulares, usando  $O(\lg n)$  operaciones de bits.

#### Teorema 31.36 (Corrección de RSA)

Las ecuaciones RSA (31.37) y (31.38) definen transformaciones inversas de  $Z_n$  que satisfacen las ecuaciones (31.35) y (31.36).

Prueba De las ecuaciones (31.37) y (31.38), tenemos que para cualquier  $M \equiv 2 \pmod{n}$ ,

$P \cdot M^{-1} \equiv D \cdot S^{-1} \pmod{n}$  :

Como  $e$  y  $d$  son inversos multiplicativos módulo  $n$ ,  $D \cdot e \equiv 1 \pmod{n}$ ,

$e \cdot D \equiv 1 \pmod{n}$

para algún entero  $k$ . Pero entonces, si  $M \equiv 0 \pmod{p}$ , tenemos

$$\begin{aligned} \text{Medicina} & M \cdot e \cdot D \equiv k \cdot q + 1 \pmod{n} & \cdot \text{mod} \\ & \equiv k \cdot q + 1 \pmod{p} & \pmod{p} \\ & \equiv k \cdot q + 1 \pmod{p} & \text{(por el Teorema 31.31)} \\ \text{METRO} & & \pmod{p} \end{aligned}$$

Además,  $\text{Med } M \equiv 0 \pmod{p}$  si  $M \equiv 0 \pmod{p}$ . Así,  $\text{Med } M \equiv 0 \pmod{p}$

para todo  $M$ . Del mismo modo,

$\text{Med } M \equiv 1 \pmod{q}$

para todo  $M$ . Así, por el Corolario 31.29 del teorema chino del resto,

$\text{Med } M \equiv 0 \pmod{n}$

para todo  $m$

■

La seguridad del criptosistema RSA se basa en gran parte en la dificultad de factorizar números enteros grandes. Si un adversario puede factorizar el módulo  $n$  en una clave pública, entonces el adversario puede derivar la clave secreta de la clave pública, usando el conocimiento de los factores  $p$  y  $q$  de la misma manera que los usó el creador de la clave pública. Por lo tanto, si factorizar números enteros grandes es fácil, romper el sistema criptográfico RSA es fácil. La afirmación contraria, que si factorizar enteros grandes es difícil, entonces descifrar RSA es difícil, no está probada. Sin embargo, después de dos décadas de investigación, no se ha encontrado un método más sencillo para descifrar el criptosistema de clave pública RSA que factorizar el módulo  $n$ . Y como veremos en la Sección 31.9, factorizar enteros grandes es sorprendentemente difícil. Al seleccionar y multiplicar al azar dos números primos de 1024 bits, podemos crear una clave pública que no se puede "romper" en ningún período de tiempo factible con la tecnología actual. En ausencia de un avance fundamental en el diseño de algoritmos teóricos de números, y cuando se implementa con cuidado siguiendo los estándares recomendados, el criptosistema RSA es capaz de proporcionar un alto grado de seguridad en las aplicaciones.

Sin embargo, para lograr la seguridad con el sistema criptográfico RSA, debemos usar números enteros que sean bastante largos: cientos o incluso más de mil bits.

largo—para resistir posibles avances en el arte del factoring. En el momento de escribir este artículo (2009), los módulos RSA se encontraban comúnmente en el rango de 768 a 2048 bits. Para crear módulos de tales tamaños, debemos ser capaces de encontrar números primos grandes de manera eficiente. La Sección 31.8 aborda este problema.

Para mayor eficiencia, RSA se usa a menudo en un modo "híbrido" o "administración de claves" con criptosistemas rápidos de clave no pública. Con dicho sistema, las claves de cifrado y descifrado son idénticas. Si Alice desea enviar un mensaje largo  $M$  a Bob en privado, selecciona una clave aleatoria  $K$  para el criptosistema rápido de clave no pública y cifra  $M$  usando  $K$ , obteniendo el texto cifrado  $C$ . Aquí,  $C$  es tan largo como  $M$ , pero  $K$  es bastante corto. Luego, encripta  $K$  usando la clave RSA pública de Bob. Dado que  $K$  es corto, calcular  $PB \cdot K$  es rápido (mucho más rápido que calcular  $PB \cdot M$ ). Luego transmite  $C; PB \cdot K$  a Bob, quien descifra  $PB \cdot K$  para obtener  $K$  y luego usa  $K$  para descifrar  $C$ , obteniendo  $M$ .

Podemos usar un enfoque híbrido similar para hacer firmas digitales de manera eficiente. Este enfoque combina RSA con una función hash pública resistente a colisiones  $h$ , una función que es fácil de calcular pero para la cual es computacionalmente inviable encontrar dos mensajes  $M$  y  $M_0$  tales que  $hM = hM_0$ . El valor  $hM$  es una "huella digital" corta (por ejemplo, de 256 bits) del mensaje  $M$ . Si Alice desea firmar un mensaje  $M$ , primero aplica  $h$  a  $M$  para obtener la huella digital  $hM$ , que luego cifra con su llave secreta. Ella envía  $.M; SA.hM$  a Bob como su versión firmada de  $M$ . Bob puede verificar la firma calculando  $hM$  y verificando que  $PA$  aplicado a  $SA.hM$  es igual a  $hM$ . Debido a que nadie puede crear dos mensajes con la misma huella dactilar, es computacionalmente inviable alterar un mensaje firmado y preservar la validez de la firma.

Finalmente, notamos que el uso de certificados facilita mucho la distribución de claves públicas. Por ejemplo, suponga que hay una "autoridad de confianza"  $T$  cuya clave pública es conocida por todos. Alice puede obtener de  $T$  un mensaje firmado (su certificado) que indique que "la clave pública de Alice es  $PA$ ". Este certificado es de "autenticación automática" ya que todo el mundo conoce  $PT$ . Alice puede incluir su certificado con sus mensajes firmados, de modo que el destinatario tenga la clave pública de Alice inmediatamente disponible para verificar su firma. Debido a que su clave fue firmada por  $T$ , la clave, que el destinatario sabe que la clave pública de Alice es realmente de Alice.

## Ejercicios

### 31.7-1

Considere un conjunto de claves RSA con  $p = 11$ ,  $q = 29$ ,  $n = 319$  y  $e = 3$ . ¿Qué valor de  $d$  debe usarse en la clave secreta? ¿Cuál es el cifrado del mensaje MD 100?

## 31.7-2

Demuestre que si el exponente público  $e$  de Alice es 3 y un adversario obtiene el exponente secreto  $d$  de Alice, donde  $0 < d < \lfloor n/3 \rfloor$ , entonces el adversario puede factorizar el módulo  $n$  de Alice en el polinomio de tiempo en el número de bits en  $n$ . (Aunque no se le pide que lo demuestre, puede interesarle saber que este resultado sigue siendo cierto incluso si se elimina la condición  $e \equiv 3 \pmod{4}$ . Véase Miller [255].)

## 31.7-3 ?

Demuestre que RSA es multiplicativo en el sentido de que

$\text{PA.M1}/\text{PA.M2}/ \text{PA.M1M2}/ \text{PA.M1}^k \pmod{n}$  :

Use este hecho para demostrar que si un adversario tuviera un procedimiento que pudiera descifrar eficientemente el 1 por ciento de los mensajes de  $Z_n$  cifrados con PA, entonces podría emplear un algoritmo probabilístico para descifrar cada mensaje cifrado con PA con alta probabilidad.

## ? 31.8 Pruebas de primalidad

En esta sección, consideramos el problema de encontrar números primos grandes. Comenzamos con una discusión sobre la densidad de los primos, procedemos a examinar un enfoque plausible, pero incompleto, para la prueba de primalidad, y luego presentamos una prueba de primalidad aleatoria efectiva de Miller y Rabin.

## La densidad de los números primos

Para muchas aplicaciones, como la criptografía, necesitamos encontrar grandes números primos "aleatorios". Afortunadamente, los números primos grandes no son demasiado raros, por lo que es factible probar números enteros aleatorios del tamaño apropiado hasta que encontremos un número primo. La función de distribución de primos  $\pi(n)$  especifica el número de primos que son menores o iguales que  $n$ . Por ejemplo,  $\pi(10) = 4$ , ya que hay 4 números primos menores o iguales que 10, a saber, 2, 3, 5 y 7. El teorema de los números primos proporciona una aproximación útil a  $\pi(n)$ .

Teorema 31.37 (Teorema de los números primos).  $\pi(n) \approx \frac{n}{\ln n}$

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{\frac{n}{\ln n}} = 1$$

La aproximación  $\pi(n) \approx \frac{n}{\ln n}$  proporciona estimaciones razonablemente precisas de  $\pi(n)$  incluso para  $n$  pequeño. Por ejemplo, tiene un error de menos del 6 % en  $\pi(109)$ , donde  $\pi(109) \approx 22$ .

50.847.534 y  $n = n$  48.254.942. (Para un teórico de números, 109 es un número pequeño).

Podemos ver el proceso de seleccionar aleatoriamente un número entero  $n$  y determinar si es primo como un ensayo de Bernoulli (consulte la Sección C.4). Por el teorema de los números primos, la probabilidad de éxito, es decir, la probabilidad de que  $n$  sea primo, es aproximadamente  $1 = \ln n / n$ . La distribución geométrica nos dice cuántos intentos necesitamos para obtener un éxito, y por la ecuación (C.32), el número esperado de intentos es aproximadamente  $\ln n$ . Por lo tanto, esperaríamos examinar aproximadamente  $\ln n$  enteros elegidos al azar cerca de  $n$  para encontrar un número primo que tenga la misma longitud que  $n$ .

Por ejemplo, esperamos que encontrar un número primo de 1024 bits requiera probar la primalidad de aproximadamente  $\ln 2^{1024} \approx 710$  números de 1024 bits elegidos al azar. (Por supuesto, podemos reducir esta cifra a la mitad eligiendo solo números enteros impares).

En el resto de esta sección, consideraremos el problema de determinar si un entero impar grande  $n$  es primo o no. Por conveniencia notacional, asumimos que  $n$  tiene la descomposición en factores primos

$$n = D \cdot p_1^{e_1} \cdot p_2^{e_2} \cdots p_r^{e_r} \quad (31.39)$$

donde  $r = 1, p_1, p_2, \dots, p_r$  son los factores primos de  $n$ , y  $e_1, e_2, \dots, e_r$  son positivos enteros. El entero  $n$  es primo si y solo si  $r = 1$  y  $e_1 = 1$ .

Un enfoque simple para el problema de las pruebas de primalidad es la división de prueba. Intentamos dividir  $n$  por cada número entero  $2, 3, \dots, \sqrt{n}$ . (Nuevamente, podemos omitir incluso los enteros mayores que  $2$ .) Es fácil ver que  $n$  es primo si y solo si ninguno de los divisores de prueba divide a  $n$ . Suponiendo que cada división de prueba toma un tiempo constante, el tiempo de ejecución del peor de los casos es  $\sqrt{n}$ , que es exponencial en la longitud de  $n$ . (Recuerde que si  $n$  se codifica en binario usando  $\lceil \log_2 n \rceil$  bits, entonces  $\sqrt{n} \leq 2^{\lceil \log_2 n \rceil}$ , y así  $\sqrt{n} \leq 2^{\lceil \log_2 n \rceil}$ .) Por lo tanto, la división de prueba funciona bien solo si  $n$  es muy pequeño o tiene un factor primo pequeño. Cuando funciona, la división de prueba tiene la ventaja de que no solo determina si  $n$  es primo o compuesto, sino que también determina uno de los factores primos de  $n$  si  $n$  es compuesto.

En esta sección sólo nos interesa averiguar si un número dado  $n$  es primo; si  $n$  es compuesto, no nos interesa encontrar su descomposición en factores primos. Como veremos en la Sección 31.9, calcular la descomposición en factores primos de un número es computacionalmente costoso. Quizás sea sorprendente que sea mucho más fácil saber si un número dado es primo o no que determinar la descomposición en factores primos del número si no es primo.

### Pruebas de pseudoprinimalidad

Ahora consideraremos un método para la prueba de primalidad que "casi funciona" y, de hecho, es lo suficientemente bueno para muchas aplicaciones prácticas. Más adelante presentaremos una re-

refinamiento de este método que elimina el pequeño defecto. Deje que  $ZC$  denote los elementos distintos de cero de  $Z_n$ :

$ZC \rightarrow f1; 2; : : : ; n \ 1g :$

Si  $n$  es primo, entonces  $ZC \rightarrow DZ$

Decimos que  $n$  es una base-un pseudoprimo si  $n$  es compuesto y

$an1\ 1 .mod\ n/ :$

(31.40)

El teorema de Fermat (Teorema 31.31) implica que si  $n$  es primo, entonces  $n$  satisface la ecuación (31.40) para todo  $a$  en  $ZC$ . Por lo tanto, si podemos encontrar cualquier  $a$  en  $ZC$  tal que  $n$  no satisface la ecuación (31.40), entonces  $n$  ciertamente es compuesto. Sorprendentemente, casi se cumple lo contrario, de modo que este criterio forma una prueba casi perfecta de primalidad.

Probamos para ver si  $n$  satisface la ecuación (31.40) para un  $D$  2. Si no, declaramos que  $n$  es compuesto devolviendo COMPOSITE. De lo contrario, devolvemos PRIME, suponiendo que  $n$  es primo (cuando, de hecho, todo lo que sabemos es que  $n$  es primo o un pseudoprimo de base 2).

El siguiente procedimiento pretende de esta manera estar comprobando la primalidad del  $n$ . Utiliza el procedimiento EXPONENCIACIÓN MODULAR de la Sección 31.6. Suponemos que la entrada  $n$  es un entero impar mayor que 2.

PSEUDOPRIME. $n/ 1$

```
si MODULAR-EXPONENTIATION.2;  $n\ 1;$   $n\ / 6\ 1 .mod\ n/$  // definitivamente //
2           volver COMPUESTO      ¡esperamos!
3 más devuelve PRIME
```

Este procedimiento puede cometer errores, pero sólo de un tipo. Es decir, si dice que  $n$  es compuesto, entonces siempre es correcto. Sin embargo, si dice que  $n$  es primo, comete un error solo si  $n$  es un pseudoprimo de base 2.

¿Con qué frecuencia falla este procedimiento? Sorprendentemente rara vez. Solo hay 22 valores de  $n$  menores de 10.000 para los que falla; los primeros cuatro de esos valores son 341, 561, 645 y 1105. No lo demostraremos, pero la probabilidad de que este programa cometa un error en un número de bit elegido al azar se reduce a cero cuando  $\sim ! 1$ . Usando estimaciones más precisas debido a Pomerance [279] del número de pseudoprimos en base 2 de un tamaño dado, podemos estimar que un número de 512 bits elegido al azar que se llama primo por el procedimiento anterior tiene menos de una posibilidad en 1020 de ser un pseudoprimo de base 2, y un número de 1024 bits elegido al azar que se llama primo tiene menos de una posibilidad en 1041 de ser un pseudoprimo de base 2. Entonces, si simplemente está tratando de encontrar un primo grande para alguna aplicación, a todos los efectos prácticos, casi nunca se equivocará al elegir números grandes al azar hasta que uno de ellos haga que PSEUDOPRIME devuelva PRIME. Pero cuando los números que se prueban para la primalidad no se eligen al azar, necesitamos un mejor enfoque para probar la primalidad.

Como veremos, un poco más de inteligencia y algo de aleatorización producirán una rutina de prueba de primalidad que funciona bien en todas las entradas.

Desafortunadamente, no podemos eliminar por completo todos los errores simplemente verificando la ecuación (31.40) para un segundo número base, digamos un D 3, porque existen números enteros compuestos n, conocidos como números de Carmichael, que satisfacen la ecuación (31.40) para (Notamos pero esperar  $n \neq 1$  que la ecuación (31.40) falla cuando  $\text{mcd}(a; n) > 1$ —que todo a 2 Z es, cuando a 62 Z n— demostrar que n es compuesto al encontrar tal a puede ser difícil si n tiene solo factores primos grandes). Los primeros tres números de Carmichael son 561, 1105 y 1729. Los números de Carmichael son extremadamente raros; hay, por ejemplo, sólo 255 de ellos menos de 100.000.000. El ejercicio 31.8-2 ayuda a explicar por qué son tan raros.

A continuación mostramos cómo mejorar nuestra prueba de primalidad para que no se deje engañar por los números de Carmichael.

#### La prueba de primalidad aleatoria de Miller-Rabin

La prueba de primalidad de Miller-Rabin supera los problemas de la prueba simple PSEUDO-DOPRIME con dos modificaciones:

Prueba varios valores base elegidos aleatoriamente en lugar de un solo valor base.

Mientras calcula cada exponenciación modular, busca una raíz cuadrada no trivial de 1, módulo n, durante el conjunto final de elevaciones al cuadrado. Si encuentra uno, se detiene y devuelve COMPOSITE. El corolario 31.35 de la Sección 31.6 justifica la detección de compuestos de esta manera.

A continuación se muestra el pseudocódigo para la prueba de primalidad de Miller-Rabin. La entrada  $n > 2$  es el número impar que se probará para la primalidad, y s es el número de valores base elegidos al azar de  $Z_C$  para ser juzgado. El código utiliza el generador de números aleatorios RANDOM descrito en la página 117: RANDOM.1; n / 1 devuelve un número entero elegido al azar que satisface  $1 \leq a \leq n - 1$ . El código utiliza un procedimiento auxiliar TESTIGO tal que TESTIGO.a; n / es VERDADERO si y sólo si a es un “testigo” de la composición de n, es decir, si es posible usar a para probar (de la manera que veremos) que n es compuesto. La prueba TESTIGO.a; n / es una extensión de, pero más eficaz que, la prueba

```
an1 6 1 .mod n/
```

que formó la base (usando un D 2) para PSEUDOPRIME. Primero presentamos y justificamos la construcción de WITNESS, y luego mostraremos cómo lo usamos en la prueba de primalidad de Miller-Rabin. Sea  $n = 1 \cdot 2^t u$  donde  $t$  y  $u$  son impares; es decir, la representación binaria de  $n$  es la representación binaria del entero impar  $u$  seguido exactamente de  $t$  ceros. Por lo tanto,  $an1 .au/2^t .mod n/$ , para que podamos

calcule  $a^{n-1} \bmod n$  calculando primero  $a^u \bmod n$  y luego elevando al cuadrado el resultado  $t$  veces sucesivamente.

```

TESTIGO.a; n/ 1
sean tu tales que t ≥ 2 x0 D           1, u es impar y n 1 D 2t u
EXPONENCIACIÓN MODULAR.a; tu; n/ 3 para i D 1 a t xi D
x2 4 5 6 7 si xt ≠ 1 8
    devuelve i1     modelo n
    si xi == 1 y xi1 ≠ 1 y xi1 ≠ n 1 devuelve VERDADERO

VERDADERO
9 devuelve FALSO

```

Este pseudocódigo para WITNESS calcula  $a^{n-1} \bmod n$  calculando primero el valor  $x_0 \equiv a^u \pmod{n}$  en la línea 2 y luego elevando al cuadrado el resultado  $t$  veces seguidas en el ciclo for de las líneas 3–6. Por inducción sobre  $i$ , la secuencia  $x_0, x_1, \dots, x_t$  de valores  $a^{2iu} \pmod{n}$  para  $i \in \{0, 1, \dots, t\}$ , por lo tanto  $x_t \equiv a^{(2t)u} \equiv a^{n-1} \pmod{n}$ . Sin embargo, después de la línea 4 que en calculado satisface la ecuación  $x_i$  particular  $x_t$  que la línea 4 realiza un paso de elevación al cuadrado, el bucle puede terminar antes de tiempo si las líneas 5 y 6 detectan que se acaba de descubrir una raíz cuadrada no trivial de 1. (Explicaremos estas pruebas en breve). Si es así, el algoritmo se detiene y devuelve VERDADERO. Las líneas 7 y 8 devuelven VERDADERO si el valor calculado para  $x_t \equiv a^{n-1} \pmod{n}$  no es igual a 1, al igual que el procedimiento PSEUDOPRIME devuelve COMPUERTO en este caso. La línea 9 devuelve FALSO si no hemos devuelto VERDADERO en las líneas 6 u 8.

Ahora argumentamos que si TESTIGO.a; n/ devuelve VERDADERO, entonces podemos construir una prueba de que  $n$  es compuesto usando a como testigo.

Si WITNESS devuelve VERDADERO de la línea 8, entonces ha descubierto que  $x_t \not\equiv a^{n-1} \pmod{n}$ . Sin embargo, si  $n$  es primo, tenemos por el teorema de Fermat (Teorema 31.31) que  $a^{n-1} \equiv 1 \pmod{n}$  para todo  $a \in \mathbb{Z}_n^\times$ . Por lo tanto,  $n$  no puede ser primo, y la ecuación  $a^{n-1} \not\equiv 1$  prueba este hecho.

Si WITNESS devuelve VERDADERO desde la línea 6, entonces ha descubierto que  $x_1$  es una raíz cuadrada no trivial de 1, módulo  $n$ , ya que tenemos que  $x_1^2 \equiv 1 \pmod{n}$  pero  $1 \not\equiv -1 \pmod{n}$ . El corolario xi  $\rightarrow$  31.35 establece que solo si  $n$  es compuesto puede existir una raíz cuadrada no trivial de 1 módulo  $n$ , de modo que demostrar que  $x_1$  es una raíz cuadrada no trivial de 1 módulo  $n$  demuestra que  $n$  es compuesto.

Esto completa nuestra prueba de la exactitud de TESTIGO. Si encontramos que la llamamos TESTIGO.a; n/ devuelve VERDADERO, entonces  $n$  seguramente es compuesto, y el testigo a, junto con la razón por la que el procedimiento devuelve VERDADERO (¿regresó de la línea 6 o de la línea 8?), proporciona una prueba de que  $n$  es compuesto.

En este punto, presentamos brevemente una descripción alternativa del comportamiento de WITNESS en función de la secuencia  $XD hx0; x1; \dots; xt$ , que encontraremos útil más adelante, cuando analicemos la eficiencia de la prueba de primalidad de Miller-Rabin.

Tenga en cuenta que si  $x_i \neq 1$  para algún  $0 \leq i < t$ , WITNESS podría no calcular el resto de la secuencia. Sin embargo, si lo hiciera, cada valor  $x_i C_1; x_i C_2; \dots; xt$  sería 1, y consideramos estas posiciones en la secuencia X como todos 1s. Tenemos cuatro casos:

1.  $XD h \dots di$ , donde  $d \neq 1$ : la secuencia X no termina en 1. Devuelva TRUE en la línea 8; a es un testigo de la composición de n (por el teorema de Fermat).
2.  $XD h1; 1; \dots; 1i$ : la secuencia X es todo 1s. Devuelve FALSO; a no es un testigo a la composición de n.
3.  $XD h \dots; 1; 1; \dots; 1i$ : la secuencia X termina en 1, y el último distinto de 1 es igual a 1. Devuelve FALSO; a no es testigo de la composición de n.
4.  $XD h \dots; d; 1; \dots; 1i$ , donde  $d \neq 1$ : la secuencia X termina en 1, pero el último distinto de 1 no es 1. Devuelve VERDADERO en la línea 6; a es un testigo de la composición de n, ya que d es una raíz cuadrada no trivial de 1.

Ahora examinamos la prueba de primalidad de Miller-Rabin basada en el uso de WITNESS. De nuevo, asumimos que n es un entero impar mayor que 2.

```

MILLER-RABIN.n; s/ 1
para j D 1 a sa D
    ALEATORIO.1; n 1/ 2 si
    TESTIGO.a; n/ retorno
    4      COMPLETO          //
    5 retorno PRIME           definitivamente // casi seguro

```

El procedimiento MILLER-RABIN es una búsqueda probabilística de una prueba de que n es compuesto. El bucle principal (que comienza en la línea 1) recoge hasta s valores aleatorios de a de  $Z_C$  (línea 2). Si una de las a elegidas es un testigo de la composición de n, entonces MILLER-RABIN devuelve COMPOSITE en la línea 4. Tal resultado siempre es correcto, por la corrección de WITNESS. Si MILLER-RABIN no encuentra testigos en los juicios s, entonces el procedimiento asume que esto se debe a que no existen testigos y, por lo tanto, asume que n es primo. Veremos que es probable que este resultado sea correcto si s es lo suficientemente grande, pero que todavía hay una pequeña posibilidad de que el procedimiento tenga mala suerte en la elección de a y que existan testigos aunque no se haya encontrado ninguno.

Para ilustrar la operación de MILLER-RABIN, sea n el número de Carmichael 561, de modo que  $n \equiv 1 \pmod{560}$ ,  $t = 4$  y  $u = 35$ . Si el procedimiento elige un  $D = 7$  como base, la Figura 31.4 en la Sección 31.6 muestra que  $WIT(241, 561)$  y por lo tanto calcula la secuencia

XD h241; 298; 166; 67; 1i. Por lo tanto, WITNESS descubre una raíz cuadrada no trivial de 1 en el último paso de elevar al cuadrado, ya que a280 67 .mod n/ y a560 1 .mod n/.

Por tanto, a D 7 es testigo de la composición de n, TESTIGO.7; n/ devuelve VERDADERO y MILLER-RABIN devuelve COMPUESTO.

Si n es un número de  $\lceil \log_2 n \rceil$  bits, MILLER-RABIN requiere operaciones aritméticas  $O(\lceil \log_2 n \rceil)$  y operaciones  $O(\lceil \log_2 n \rceil^3)$  de exponentes modulares.

#### Tasa de error de la prueba de primalidad de Miller-Rabin

Si MILLER-RABIN devuelve PRIME, entonces hay una posibilidad muy pequeña de que haya cometido un error. Sin embargo, a diferencia de PSEUDOPRIME, la posibilidad de error no depende de n; no hay malas entradas para este procedimiento. Más bien, depende del tamaño de s y de la "suerte del sorteo" al elegir los valores base a. Además, dado que cada prueba es más estricta que una simple verificación de la ecuación (31.40), podemos esperar, según los principios generales, que la tasa de error sea pequeña para los enteros n elegidos al azar. El siguiente teorema presenta un argumento más preciso.

#### Teorema 31.38 Si n

es un número compuesto impar, entonces el número de testigos de la composición de n es al menos  $\frac{1}{4} \ln n$ .

Prueba La prueba muestra que el número de no testigos es como mucho  $\frac{1}{4} \ln n$ , lo que implica el teorema.

Comenzamos afirmando que cualquier persona que no sea testigo debe ser miembro de  $Z_n^*$ . ¿Por qué? Considere cualquier no testigo a. Debe satisfacer  $a^{n-1} \equiv 1 \pmod{n}$ , de manera equivalente,  $a^n \equiv a \pmod{n}$ . Por lo tanto, la ecuación  $a^x \equiv 1 \pmod{n}$  tiene una solución, a saber,  $x=n$ . Por el Corolario 31.21,  $\gcd(a; n) = 1$ , lo que a su vez implica que  $\gcd(a; n) = 1$ . Por lo tanto, a es miembro de  $Z_n^*$ ; todos los no testigos pertenecen a  $Z_n^*$ .

Para completar la prueba, mostramos que no solo están contenidos todos los no testigos (recuerde que en  $Z_n^*$  todos están contenidos en un subgrupo propio B de  $Z_n^*$ ) decimos que B es un subgrupo propio de  $Z_n^*$  cuando B es un subgrupo de  $Z_n^*$  pero B no es igual a  $Z_n^*$ .

Por el Corolario 31.16, tenemos  $jBj \mid jZ_n^*j = \phi(n)$ . Como  $jZ_n^*j \leq n-1$ , obtenemos  $jBj \geq \frac{n-1}{\phi(n)}$ . Por tanto, el número de no testigos es como máximo  $\frac{n-1}{\phi(n)}$ , por lo que el número de testigos debe ser como mínimo  $\frac{\phi(n)}{2}$ .

Ahora mostramos cómo encontrar un subgrupo B propio de  $Z_n^*$  que contiene todos los no testigos. Dividimos la demostración en dos casos. tal que

Caso 1: Existe un  $x \in Z_n^*$

$x^{n-1} \not\equiv 1 \pmod{n}$ :

En otras palabras,  $n$  no es un número de Carmichael. Debido a que, como señalamos anteriormente, los números de Carmichael son extremadamente raros, el caso 1 es el caso principal que surge "en la práctica" (p. ej., cuando  $n$  se ha elegido al azar y se está probando su primalidad).

Sea  $B \subset \mathbb{Z}^*$   $\text{mod } n$ . Claramente,  $B$  no es vacío, ya que  $1 \in B$ . Como  $B$  es cerrado bajo la multiplicación módulo  $n$ , tenemos que  $B$  es un subgrupo de  $\mathbb{Z}^*$  por el teorema 31.14. Nótese que todo no testigo pertenece a  $B$ , ya que un no  $B$ , tenemos que  $B$  testigo a cumple  $a \equiv 1 \pmod{n}$ . Como  $x \in \mathbb{Z}^*$  subgrupo propio  $\text{mod } n$  es un de  $\mathbb{Z}^*$

Caso 2: Para todos  $x \in \mathbb{Z}^*$

$$x \equiv 1 \pmod{n} : \quad (31.41)$$

En otras palabras,  $n$  es un número de Carmichael. Este caso es extremadamente raro en la práctica. Sin embargo, la prueba de Miller-Rabin (a diferencia de una prueba de pseudoprimalidad) puede determinar de manera eficiente que los números de Carmichael son compuestos, como mostramos ahora.

En este caso,  $n$  no puede ser una potencia prima. Para ver por qué, supongamos por el contrario que  $n = p^e$ , donde  $p$  es primo y  $e > 1$ . Derivamos una contradicción de la siguiente manera. Dado que suponemos que  $n$  es impar,  $p$  también debe ser impar. El teorema 31.32 implica que  $\mathbb{Z}^*$  es un grupo cíclico: contiene un generador  $g$  tal que  $ord_g \equiv D \pmod{p^e}$ . La fórmula para  $ord_g$  proviene de la ecuación (31.20). Por la ecuación (31.41), tenemos  $g \equiv 1 \pmod{n}$ . Entonces el teorema del logaritmo discreto (Teorema 31.33, tomando  $y \equiv 1 \pmod{n}$ ) implica que  $g \equiv 1 \pmod{p^e}$ , o

$$\frac{1}{p^{e-1}} \equiv 1 \pmod{p}$$

Esto es una contradicción para  $e > 1$ , ya que  $\frac{1}{p^{e-1}}$  es divisible por el primo  $p$  pero  $p$  no lo es. Por lo tanto,  $n$  no es una potencia prima.

Dado que el número compuesto impar  $n$  no es una potencia prima, lo descomponemos en un producto  $n_1 n_2$ , donde  $n_1$  y  $n_2$  son números impares mayores que 1 que son primos entre sí. (Puede haber varias formas de descomponer  $n$ , y no importa cuál elijamos. Por ejemplo, si  $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$  entonces podemos elegir  $n_1 = p_1^{e_1} p_2^{e_2} \dots p_{k-1}^{e_{k-1}}$  y  $n_2 = p_k^{e_k}$ ).

1                    2                    3                     $r$

Recuerde que definimos  $t$  y  $u$  de modo que  $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k} = t^r u$ , donde  $t$  y  $u$  son impares, y que para una entrada  $a$ , el procedimiento WITNESS calcula la secuencia

jaja XD ; a2u; a22u;:::;a2tui

(Todos los cálculos se realizan módulo  $n$ ).

Llamemos a un par.  $(j, t)$  de enteros aceptable si  $2^j \equiv 1 \pmod{n}$  y  $2^t \equiv 1 \pmod{n}$ .

Ciertamente existen pares aceptables ya que  $u$  es impar; podemos elegir  $D \equiv 1 \pmod{4}$ , de modo que  $n \equiv 0 \pmod{D}$  es un par aceptable. Ahora escoja el  $j$  más grande posible tal que exista un par aceptable  $u \equiv j^2 \pmod{D}$ , y arreglar para que  $u \equiv j^2 \pmod{D}$  es un par aceptable. Dejar

$$BD \equiv x^2 \pmod{Z} \quad u \equiv x^2 \pmod{D} \quad u \equiv j^2 \pmod{D}$$

Como  $B$  es cerrado bajo el módulo de multiplicación  $n$ , es un subgrupo de  $Z$ . Por el teorema 31.15, por lo tanto,  $\text{gcd}(B, D) = 1$ . Todo no testigo debe ser miembro de  $B$ , ya que la secuencia  $X$  producida por un no testigo debe ser todo 1 o contener un 1 a más tardar en la  $j$ -ésima posición, por la maximalidad de  $j$ . (Si  $a \equiv j^2 \pmod{D}$  es aceptable, donde  $a \neq 1$  es un testigo, debemos tener  $j > n/2$  por cómo elegimos  $j$ .)

Ahora usamos la existencia de para demostrar que existe  $w \in Z$  tal que  $w^2 \equiv j^2 \pmod{D}$ , y por lo tanto que  $B$  es un subgrupo propio de  $Z$ . Desde  $w^2 \equiv j^2 \pmod{D}$ , tenemos  $w \equiv j \pmod{D}$  por el Corolario 31.29 del teorema chino del resto. Por

Corolario 31.28, existe  $w$  que satisface simultáneamente las ecuaciones

$$w \equiv u \pmod{n_1};$$

$$w \equiv 1 \pmod{n_2};$$

Por lo tanto,  $w \equiv j \pmod{D}$

$$u \equiv 1 \pmod{n_1}; w \equiv 1 \pmod{n_2}; \text{ Por el}$$

Corolario 31.29,  $w \equiv j \pmod{D}$  implica

$w^2 \equiv j^2 \pmod{D}$ , y  $w^2 \equiv 1 \pmod{D}$  implica  $w \equiv \pm 1 \pmod{D}$ . Por lo tanto, concluimos que  $w \equiv \pm 1 \pmod{D}$ , por lo que  $w \equiv 1 \pmod{D}$ .

Queda por demostrar que  $w \in B$ . Lo que hacemos trabajando primero por separado modulo  $n_1$  y módulo  $n_2$ . Trabajando módulo  $n_1$ , observamos que dado que  $w \in B$  tienen ese  $\text{mcd}(w, n_1) = 1$ , y así también  $\text{mcd}(w^2, n_1) = 1$ ; si no tiene divisores comunes con  $n_1$ , entonces ciertamente no tiene divisores comunes con  $n_1$ .  $w \equiv 1 \pmod{n_1}$ , vemos que  $\text{mcd}(w, n_1) = 1$ . Trabajando módulo  $n_2$ , estos resultados implican que  $w \equiv 1 \pmod{n_2}$ . Como  $w \equiv 1 \pmod{n_2}$  implica  $\text{mcd}(w, n_2) = 1$ . Para combinar resultados, usamos el Teorema 31.6, que implica que  $\text{mcd}(w, n_1n_2) = \text{mcd}(w, n_1)\text{mcd}(w, n_2) = 1$ . Es decir,

$$w \in Z$$

Por lo tanto  $w \in B$ , y terminamos el caso 2 con la conclusión de que  $B$  es un subgrupo propio de  $Z$ .

En cualquier caso, vemos que el número de testigos de la composición de  $n$  es al menos  $n^{1/2}$ . ■

### Teorema 31.39

Para cualquier entero impar  $n > 2$  y entero positivo  $s$ , la probabilidad de que MILLER RABIN  $n$  sea伪 (falso) es como mucho  $2^{-s}$ .

Demostración Usando el teorema 31.38, vemos que si  $n$  es compuesto, entonces cada ejecución del ciclo for de las líneas 1–4 tiene una probabilidad de al menos  $1=2$  de descubrir un testigo  $x$  de la composición de  $n$ . MILLER-RABIN comete un error solo si tiene la mala suerte de no descubrir un testigo de la composición de  $n$  en cada una de las iteraciones del bucle principal. La probabilidad de tal secuencia de fallas es como mucho  $2s$  ■

Si  $n$  es primo, MILLER-RABIN siempre reporta PRIMO, y si  $n$  es compuesto, la probabilidad de que MILLER-RABIN reporte PRIMO es como

mucho  $2s$  Cuando aplicamos MILLER-RABIN a un entero grande  $n$  elegido al azar, sin embargo, necesitamos considerar como bien la probabilidad previa de que  $n$  sea primo, para interpretar correctamente el resultado de MILLER-RABIN. Supongamos que fijamos una longitud de bit  $\gamma$  y elegimos al azar un número entero  $n$  de longitud  $\gamma$  bits para probar la primalidad. Sea  $A$  el evento de que  $n$  es primo. Por el teorema de los números primos (Teorema 31.37), la probabilidad de que  $n$  sea primo es aproximadamente

$$\Pr_{\text{fag}} \quad 1 = \ln n \\ n^{1:443} = \gamma :$$

Ahora permita que  $B$  denote el evento de que MILLER-RABIN devuelve PRIME. tenemos eso

$\Pr^{\circ} B \mid AD 0$  (o de manera equivalente, que  $\Pr fB \mid Ag D 1$ ) y  $\Pr^{\circ} B \mid A 2s$  (o de manera equivalente, que  $\Pr^{\circ} B \mid A > 1 2s$ ).

Pero, ¿cuál es  $\Pr fA \mid Bg$ , la probabilidad de que  $n$  sea primo, dado que MILLER RABIN ha devuelto PRIMO? Por la forma alternativa del teorema de Bayes (ecuación (C.18)) tenemos

$$\Pr fA \mid Bg D \quad \frac{\Pr fAg \Pr fB \mid Ag}{\Pr fAg \Pr fB \mid Ag C \Pr^{\circ} A \Pr^{\circ} B \mid A} \\ \frac{1}{1 C 2s . \ln n 1 /}$$

Esta probabilidad no excede  $1=2$  hasta que  $s$  excede  $\lg \ln n 1/$ . Intuitivamente, se necesitan muchas pruebas iniciales solo por la confianza derivada de no encontrar un testigo de la composición de  $n$  para superar el sesgo anterior a favor de que  $n$  sea compuesto. Para un número con  $\gamma$  D 1024 bits, esta prueba inicial requiere aproximadamente

$$\lg \ln n 1 / \quad \lg \gamma = 1:443 / \\ 9$$

juicios En cualquier caso, elegir  $s$  D 50 debería ser suficiente para casi cualquier aplicación imaginable.

De hecho, la situación es mucho mejor. Si estamos tratando de encontrar números primos grandes aplicando MILLER-RABIN a enteros impares grandes elegidos al azar, entonces es muy poco probable que elegir un valor pequeño de  $s$  (digamos 3) conduzca a resultados erróneos, aunque

No lo probaremos aquí. La razón es que para un entero compuesto impar  $n$  elegido al azar, el número esperado de personas que no son testigos de la composición de  $n$  probablemente sea mucho menor que  $\frac{n}{1} = 2$ .

Sin embargo, si el número entero  $n$  no se elige al azar, lo mejor que se puede probar es que el número de no testigos es como máximo  $\frac{n}{1} = 4$ , usando una versión mejorada del Teorema 31.38. Además, existen números enteros  $n$  para los cuales el número de no testigos es  $\frac{n}{1} = 4$ .

### Ejercicios

#### 31.8-1

Demuestre que si un entero impar  $n > 1$  no es un primo ni una potencia prima, entonces existe una raíz cuadrada no trivial de 1 módulo  $n$ .

#### 31.8-2 ?

Es posible reforzar ligeramente el teorema de Euler a la forma

$$a^n \equiv 1 \pmod{n} \text{ para todos } a \in \mathbb{Z}^{\text{norte}},$$

donde  $n \neq 1$  y  $a^n \equiv 1 \pmod{n}$  está definido por

$$\text{norte} = \text{lcm}(a-1, n) \quad (31.42)$$

Demostrar que  $n \neq 1$  y  $a^n \equiv 1 \pmod{n}$ . Un número compuesto  $n$  es un número de Carmichael si  $\text{lcm}(n-1, n) = n$ . El número de Carmichael más pequeño es 561 ( $3 \cdot 11 \cdot 17$ ; aquí,  $\text{lcm}(2, 10, 16) = 80$ , que divide a 560). Demuestre que los números de Carmichael deben ser tanto "libres de cuadrados" (no divisibles por el cuadrado de ningún primo) como el producto de al menos tres primos. (Por esta razón, no son muy comunes).

#### 31.8-3

Demuestre que si  $x$  es una raíz cuadrada no trivial de 1, módulo  $n$ , entonces  $\text{mcd}(x-1, n) \neq 1$  y  $\text{mcd}(x+1, n) \neq 1$ ;  $n$  son ambos divisores no triviales de  $n$ .

## ? 31.9 Factorización de enteros

Supongamos que tenemos un número entero  $n$  que deseamos factorizar, es decir, descomponerlo en un producto de números primos. La prueba de primalidad de la sección anterior puede decirnos que  $n$  es compuesto, pero no nos dice los factores primos de  $n$ . Factorizar un entero grande  $n$  parece ser mucho más difícil que simplemente determinar si  $n$  es primo o compuesto. Incluso con las supercomputadoras de hoy y los mejores algoritmos hasta la fecha, no podemos factorizar de manera factible un número arbitrario de 1024 bits.

## Heurística rho de Pollard

Se garantiza que la división de prueba por todos los números enteros hasta R factorizará completamente cualquier número hasta R2. Para la misma cantidad de trabajo, el siguiente procedimiento, POLLARD-RHO, factoriza cualquier número hasta R4 (a menos que tengamos mala suerte). Dado que el procedimiento es solo una heurística, no se garantiza ni su tiempo de ejecución ni su éxito, aunque el procedimiento es altamente efectivo en la práctica. Otra ventaja del procedimiento POLLARD RHO es que utiliza solo un número constante de ubicaciones de memoria. (Si quisiera, podría implementar fácilmente POLLARD-RHO en una calculadora de bolsillo programable para encontrar factores de números pequeños).

POLLARD-RHO.n/

```

1 i D 1 2
x1 D ALEATORIO.0; n 1/ 3 y D x1
4 k D 2 5
mientras
que VERDADERO
6 i D i C 1 xi D .x2
7           i1      1/ mod nd
8       D gcd.y xi; n/ si d ≠ 1 y
9       d ≠ n imprime d si i ==
10          k
11
12          y D xi k
13          D 2k

```

El procedimiento funciona de la siguiente manera. Las líneas 1 y 2 inicializan i en 1 y x1 en un valor elegido al azar en  $Z_n$ . El ciclo while que comienza en la línea 5 itera eternamente, buscando factores de n. Durante cada iteración del ciclo while , la línea 7 usa la recurrencia xi

$$D .x2 \quad i1 \quad 1/ \text{mod} \ n \quad (31.43)$$

para producir el siguiente valor de xi en la secuencia infinita

$$x1; x2; x3; x4; \dots; \quad (31.44)$$

con la línea 6 incrementando correspondientemente i. El pseudocódigo se escribe utilizando variables con subguiones xi para mayor claridad, pero el programa funciona igual si se descartan todos los subguiones, ya que solo es necesario mantener el valor más reciente de xi . Con esta modificación, el procedimiento usa solo un número constante de ubicaciones de memoria.

De vez en cuando, el programa guarda el valor xi generado más recientemente en la variable y. En concreto, los valores que se guardan son aquellos cuyos subíndices son potencias de 2:

x1; x2; x4; x8; x16;::: :

La línea 3 guarda el valor  $x_1$  y la línea 12 guarda  $x_k$  siempre que  $i$  sea igual a  $k$ . La variable  $k$  se inicializa a 2 en la línea 4, y la línea 13 la duplica cada vez que la línea 12 actualiza  $y$ . Por lo tanto,  $k$  sigue la secuencia 1; 2; 4; 8; :: : y siempre da el subíndice del siguiente valor  $x_k$  a guardar en  $y$ .

Las líneas 8 a 10 tratan de encontrar un factor de  $n$ , utilizando el valor guardado de  $y$  y el valor actual de  $x_i$ . Específicamente, la línea 8 calcula el máximo común divisor d mcd.y xi; norte/. Si la línea 9 encuentra que  $d$  es un divisor no trivial de  $n$ , entonces la línea 10 imprime  $d$ .

Este procedimiento para encontrar un factor puede parecer algo misterioso al principio. Tenga en cuenta, sin embargo, que POLLARD-RHO nunca imprime una respuesta incorrecta; cualquier número que imprima es un divisor no trivial de  $n$ . Sin embargo, POLLARD-RHO podría no imprimir nada en absoluto; viene sin garantía de que imprimirá ningún divisor. Veremos, sin embargo, que tenemos buenas razones para esperar que POLLARD-RHO imprima un factor  $p$  de  $n$  después de ..pp/ iteraciones del ciclo while . Por lo tanto, si  $n$  es compuesto, podemos esperar que este procedimiento descubra suficientes divisores para factorizar  $n$  completamente después de aproximadamente  $n^{1/4}$  actualizaciones, ya que cada factor primo  $p$  de  $n$ , excepto posiblemente el más grande, es menor que  $\sqrt{n}$ .

Comenzamos nuestro análisis de cómo se comporta este procedimiento estudiando cuánto tarda una secuencia aleatoria módulo  $n$  en repetir un valor. Dado que  $Z_n$  es finito, y dado que cada valor de la sucesión (31.44) depende únicamente del valor anterior, la sucesión (31.44) finalmente se repite. Una vez que alcanzamos un  $x_i$  tal que  $x_i \equiv x_j \pmod{n}$  para algún  $j < i$ , estamos en un ciclo, ya que  $x_{i+1} \equiv x_{j+1} \pmod{n}$ ,  $x_{i+2} \equiv x_{j+2} \pmod{n}$ , y así sucesivamente.

La razón del nombre "heurística rho" es que, como muestra la figura 31.7, podemos dibujar la secuencia  $x_1; x_2; \dots; x_{i-1}$  como la "cola" de la rho y el ciclo  $x_i; x_{i+1}; \dots; x_{i+k}$  como el "cuerpo" de la rho.

Consideraremos la cuestión de cuánto tarda en repetirse la secuencia de  $x_i$ .

Esta información no es exactamente la que necesitamos, pero veremos más adelante cómo modificar el argumento. Para el propósito de esta estimación, supongamos que la función

$f_n(x) = x^2 \pmod{n}$

se comporta como una función "aleatoria". Por supuesto, no es realmente aleatorio, pero esta suposición produce resultados consistentes con el comportamiento observado de POLLARD-RHO. Entonces podemos considerar que cada  $x_i$  se extrae independientemente de  $Z_n$  de acuerdo con una distribución uniforme en  $Z_n$ . Por el análisis de la paradoja del cumpleaños de la Sección 5.4.1, esperamos que se tomen los pasos  $\sqrt{n}$  antes de los ciclos de secuencia.

Ahora para la modificación requerida. Sea  $p$  un factor no trivial de  $n$  tal que  $\text{mcd}(p, n) = p$ ;  $n = p \cdot D$ . Por ejemplo, si  $n$  tiene la factorización  $n = p_1 p_2 \dots p_k$  entonces podemos tomar  $p$  como  $p_1$  (Si  $p_1 = D$ , entonces  $p$  es solo el factor primo más pequeño de  $n$ , un buen ejemplo para tener en cuenta).

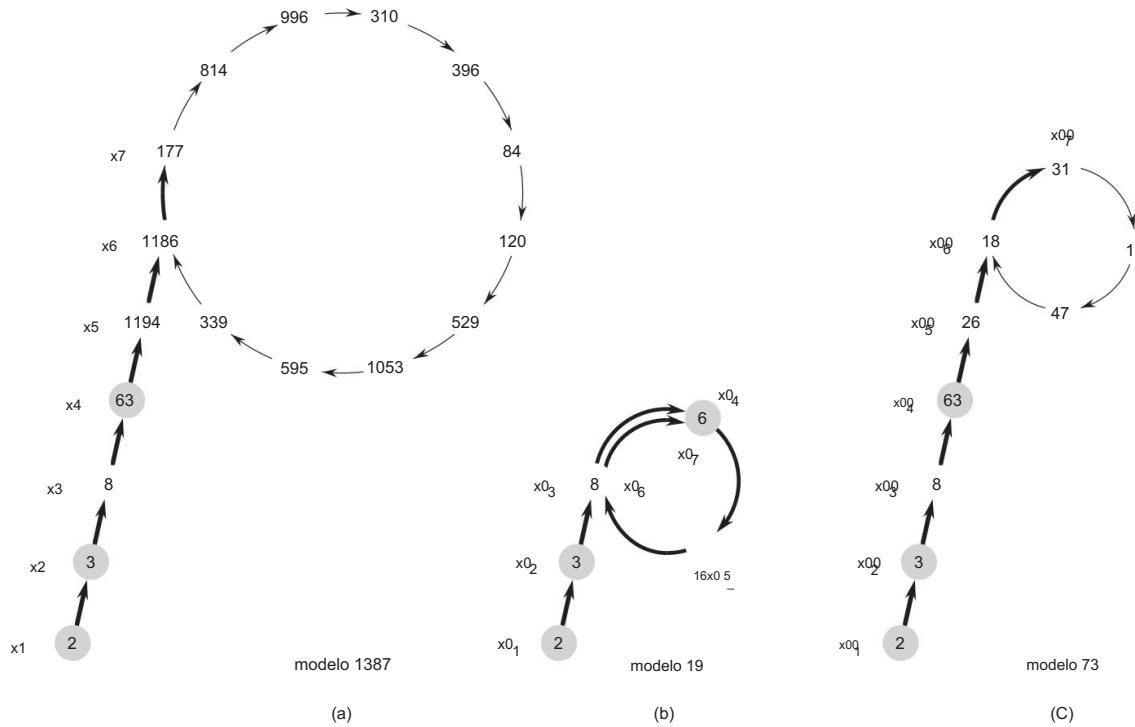


Figura 31.7 Heurística rho de Pollard. (a) Los valores producidos por la recurrencia  $x_i C_1 D .x_2 1 / \text{mod } 1387$ , comenzando con  $x_1 D 2$ . La factorización prima de 1387 es  $19 \cdot 73$ . Las flechas gruesas indican los pasos de iteración que se ejecutan antes de que se descubra el factor 19. Las flechas de luz apuntan a valores no alcanzados en la iteración, para ilustrar la forma "rho". Los valores sombreados son los valores y almacenados por POLLARD-RHO. El factor 19 se descubre al llegar a  $x_7 D 177$ , cuando  $\text{mcd}.63 \cdot 177; 1387 / 19$  se computa. El primer valor de  $x$  que se repite es 1186, pero el factor 19 se descubre antes de que se repita este valor. (b) Los valores producidos por la misma recurrencia, módulo 19. Todo valor  $x_i$  dado en el inciso (a) es equivalente, módulo 19, al valor  $x_0$  que se muestra aquí. Por ejemplo, tanto  $x_4 D 63$  como  $x_7 D 177$  son equivalentes a 6, módulo 19. (c) Los valores producidos por la misma recurrencia, módulo 73. Todo valor  $x_i$  dado en el inciso (a) es equivalente, módulo 73, al valor  $x_0$  que se muestra aquí. Por el teorema chino del resto, cada nodo en la parte (a) corresponde a un par  $i$  de nodos, uno de la parte (b) y uno de la parte (c).

La secuencia  $hx_{ii}$  induce una secuencia correspondiente  $hx_0 ii$  módulo  $p$ , donde

$$x_i^0 D x_i \text{ mod } p$$

por todo yo

Además, debido a que  $f_n$  se define usando solo operaciones aritméticas (el cuadrado y resta) módulo  $n$ , podemos calcular  $x_0^{iC_1}$  de  $x_0 i$ ; la vista de "módulo  $p$ " de

la secuencia es una versión más pequeña de lo que está pasando modulo n:

$$\begin{aligned}
 & x_0 \mod p \\
 & \quad \vdots \\
 & \quad f_n \mod p \dots x_2 \\
 & \quad \quad \quad \vdots \quad 1 \mod n \mod p \mod 1 \\
 & \quad D \dots x_2 \quad \mod p \quad \text{(por el ejercicio 31.1-7)} \\
 & \quad D \dots x_i \mod p/2 \mod p \mod p \dots x_0 \mod 1 \\
 & \quad \mod p \mod p \mod p \dots x_0 \mod 1 : 
 \end{aligned}$$

Así, aunque no estemos calculando explícitamente la sucesión  $hx_0$ , esta sucesión está bien definida y obedece a la misma recurrencia que la sucesión  $hx_i$ .

Razonando como antes, encontramos que el número esperado de pasos antes de que se repita la secuencia  $hx_0$  es  $\lfloor \frac{n}{p} \rfloor$ . Si  $p$  es pequeño en comparación con  $n$ , la secuencia  $hx_0$  podría repetirse mucho más rápido que la secuencia  $hx_i$ . De hecho, como muestran las partes (b) y (c) de la figura 31.7, la secuencia  $hx_0$  se repite tan pronto como dos elementos de la secuencia  $hx_i$  son meramente módulo  $p$  equivalentes, en lugar de módulo  $n$  equivalentes.

Sea  $t$  el índice del primer valor repetido en la secuencia  $hx_0$ , y sea  $u > 0$  la duración del ciclo que se ha producido de ese modo. Es decir,  $t$  y  $u > 0$  son los valores más pequeños tales que  $x_0$  para todo  $i \geq 0$ . Según los argumentos  $t_{Ci}$  anteriores, los valores  $x_{t+Cu_i}$  esperados de  $t$  y  $u$  son ambos  $\dots pp$ . Tenga en cuenta que si  $x_0$  entonces  $p \mid x_{t+Cu_i} - x_t$ . Así,  $\gcd(x_{t+Cu_i} - x_t, n) > 1$ .

$$x_0 \mod t_{Cu_i},$$

Por lo tanto, una vez que POLLARD-RHO ha guardado como  $y$  cualquier valor  $x_k$  tal que  $k \geq t$ , entonces  $y \mod p$  siempre está en el ciclo módulo  $p$ . (Si se guarda un nuevo valor como  $y$ , ese valor también está en el ciclo módulo  $p$ ). Eventualmente,  $k$  se establece en un valor que es mayor que  $u$ , y el procedimiento hace un ciclo completo alrededor del ciclo módulo  $p$  sin cambiar el valor de  $y$ . Entonces, el procedimiento descubre un factor de  $n$  cuando  $x_i$  "se encuentra" con el valor de  $y$  previamente almacenado, módulo  $p$ , es decir, cuando  $x_i \equiv y \pmod{p}$ .

Presumiblemente, el factor encontrado es el factor  $p$ , aunque ocasionalmente puede suceder que se descubra un múltiplo de  $p$ . Dado que los valores esperados de  $t$  y  $u$  son  $\lfloor \frac{n}{p} \rfloor$ , el número esperado de pasos necesarios para producir el factor  $p$  es  $\lfloor \frac{n}{p} \rfloor$ .

Es posible que este algoritmo no funcione como se esperaba, por dos razones. Primero, el análisis heurístico del tiempo de ejecución no es riguroso y es posible que el ciclo de valores, módulo  $p$ , sea mucho mayor que  $\lfloor \frac{n}{p} \rfloor$ . En este caso, el algoritmo funciona correctamente pero mucho más lento de lo deseado. En la práctica, este problema parece ser discutible. En segundo lugar, los divisores de  $n$  producidos por este algoritmo siempre pueden ser uno de los factores triviales 1 o  $n$ . Por ejemplo, suponga que  $n = pq$ , donde  $p$  y  $q$  son primos. Puede suceder que los valores de  $t$  y  $u$  para  $p$  sean idénticos a los valores de  $t$  y  $u$  para  $q$ , y así el factor  $p$  siempre se revela en la misma operación gcd que revela el factor  $q$ . Dado que ambos factores se revelan al mismo

tiempo, se revela el factor trivial  $pq \mid n$ , que es inútil. Nuevamente, este problema parece ser insignificante en la práctica. Si es necesario, podemos reiniciar la heurística con una recurrencia diferente de la forma  $x_{i+1} = f(x_i) \bmod n$ . (Debemos evitar los valores  $c \equiv 0 \pmod{n}$  y  $c \equiv 2 \pmod{n}$  por razones que no abordaremos aquí, pero otros valores están bien).

Por supuesto, este análisis es heurístico y no riguroso, ya que la recurrencia no es realmente "aleatoria". No obstante, el procedimiento funciona bien en la práctica y parece ser tan eficiente como indica este análisis heurístico. Es el método de elección para encontrar factores primos pequeños de un número grande. Para factorizar completamente un número compuesto  $n$  de  $\ell$  bits, solo necesitamos encontrar todos los factores primos menores que  $b^{n-1} \equiv 1 \pmod{n}$ , por lo que esperamos que POLLARD-RHO requiera como máximo  $n^{\frac{1}{2}} = 2^{\lceil \frac{\ell}{2} \rceil}$  operaciones aritméticas y como máximo  $n^{\frac{1}{2}} = 2^{\lceil \frac{\ell}{2} \rceil} \cdot 2^{\lceil \frac{\ell}{2} \rceil}$  operaciones con bits. La capacidad de POLLARD-RHO para encontrar un pequeño factor  $p$  de  $n$  con un número esperado  $\sqrt{pp}/\ell$  de operaciones aritméticas suele ser su característica más atractiva.

### Ejercicios

#### 31.9-1

Con referencia al historial de ejecución que se muestra en la figura 31.7(a), ¿cuándo imprime POLLARD RHO el factor 73 de 1387?

#### 31.9-2

Suponga que se nos da una función  $f: \mathbb{Z}_n \rightarrow \mathbb{Z}_n$  y un valor inicial  $x_0 \in \mathbb{Z}_n$ .

Defina  $x_i = f^{(i)}(x_0)$  para  $i = 1, 2, \dots$ . Sean  $t$  y  $u > 0$  los valores más pequeños tales que  $x_t \equiv x_u \pmod{n}$ . En la terminología del algoritmo rho de Pollard,  $t$  es la longitud de la cola y  $u$  es la longitud del ciclo de la rho. Proporcione un algoritmo eficiente para determinar  $t$  y  $u$  exactamente, y analice su tiempo de ejecución.

#### 31.9-3

¿Cuántos pasos esperaría que requiriera POLLARD-RHO para descubrir un factor de la forma  $p^e$ , donde  $p$  es primo y  $e > 1$ ?

#### 31.9-4 ?

Una desventaja de POLLARD-RHO tal como está escrito es que requiere un cálculo gcd para cada paso de la recurrencia. En su lugar, podríamos procesar por lotes los cálculos de gcd acumulando el producto de varios valores  $x_i$  seguidos y luego usar este producto en lugar de  $x_i$  en el cálculo de gcd. Describa cuidadosamente cómo implementaría esta idea, por qué funciona y qué tamaño de lote elegiría como el más efectivo cuando trabaje en un número  $n$  de  $\ell$  bits.

## Problemas

### 31-1 Algoritmo gcd binario

La mayoría de las computadoras pueden realizar las operaciones de resta, probar la paridad (par o impar) de un entero binario y reducir a la mitad más rápidamente que calcular los restos.

Este problema investiga el algoritmo mcd binario, que evita los cálculos restantes utilizados en el algoritmo de Euclides.

- a. Demostrar que si  $a$  y  $b$  son ambos pares, entonces  $\text{mcd}(a; b) \leq \frac{1}{2} \text{mcd}(a/2; b/2)$ .
- b. Demostrar que si  $a$  es impar y  $b$  es par, entonces  $\text{mcd}(a; b) = \text{mcd}(a; b/2)$ .
- c. Demuestre que si  $a$  y  $b$  son ambos impares, entonces  $\text{mcd}(a; b) \leq \frac{1}{2} \text{mcd}((a+b)/2; |a-b|/2)$ .
- d. Diseñe un algoritmo gcd binario eficiente para los enteros de entrada  $a$  y  $b$ , donde  $ab$ , que se ejecuta en un tiempo  $O(\lg ab)$ . Suponga que cada resta, prueba de paridad y reducción a la mitad toma una unidad de tiempo.

### 31-2 Análisis de operaciones con bits en el algoritmo de Euclides

a. Considere el algoritmo ordinario de "papel y lápiz" para la división larga: dividir  $a$  por  $b$ , lo que produce un cociente  $q$  y un resto  $r$ . Demuestre que este método requiere operaciones con bits  $O(\lg q \lg b)$ .

- b. Defina  $\text{op}_1(a; b) = \lg a + \lg b$ . Muestre que el número de operaciones de bits realizadas por EUCLID reduce el problema de calcular  $\text{gcd}(a; b)$  a la de computar  $\text{gcd}(b; a \bmod b)$  como mucho  $\text{op}_1(a; b) - \text{op}_1(b; a \bmod b)$  para alguna constante suficientemente grande  $c > 0$ .
- c. Demuestre que EUCLID( $a; b$ ) requiere  $O(\lg a + \lg b)$  operaciones con bits en general y operaciones de  $O(2^{\lfloor \lg a \rfloor})$  bit cuando se aplican a entradas de dos  $\lceil \lg a \rceil$ -bit.

### 31-3 Tres algoritmos para números de Fibonacci

Este problema compara la eficiencia de tres métodos para calcular el  $n$ -ésimo número de Fibonacci  $F_n$ , dado  $n$ . Suponga que el costo de sumar, restar o multiplicar dos números es  $O(1)$ , independientemente del tamaño de los números.

- a. Demuestre que el tiempo de ejecución del método recursivo directo para calcular  $F_n$  basado en la recurrencia (3.22) es exponencial en  $n$ . (Ver, por ejemplo, el procedimiento FIB en la página 775.)
- b. Muestre cómo calcular  $F_n$  en  $O(n)$  time usando memorización.

C. Muestre cómo calcular  $F_n$  en  $O(\lg n)$  tiempo usando solo la suma y la multiplicación de enteros. (Sugerencia: Considere la matriz

$$\begin{matrix} 0 & 1 & 1 \\ & 1 \end{matrix}$$

y sus poderes).

d. Suponga ahora que sumar dos números de  $\lceil b \rceil$  bit lleva  $\lceil b \rceil$  tiempo y que multiplicar dos números de  $\lceil b \rceil$  bit lleva  $\lceil b^2 \rceil$  tiempo. ¿Cuál es el tiempo de ejecución de estos tres métodos bajo esta medida de costo más razonable para las operaciones aritméticas elementales?

#### 31-4 Residuos cuadráticos

Sea  $p$  un primo impar. Un número  $a \in \mathbb{Z}$  es un residuo cuadrático si la ecuación  $a \equiv x^2 \pmod{p}$  tiene una solución para la incógnita  $x$ .

- a. Muestre que hay exactamente  $\frac{p-1}{2}$  residuos cuadráticos, módulo  $p$ .
- b. Si  $p$  es primo, definimos el símbolo de Legendre  $\left(\frac{a}{p}\right)$ , para un  $a \in \mathbb{Z}_p$ , ser 1 si  $a$  es un residuo cuadrático módulo  $p$  y -1 en caso contrario. Demostrar que si  $a \in \mathbb{Z}_p$ , entonces

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \pmod{p}$$

Proporcione un algoritmo eficiente que determine si un número dado  $a$  es un residuo cuadrático módulo  $p$ . Analice la eficiencia de su algoritmo.

- C. Demostrar que si  $p$  es un primo de la forma  $4k+3$  y  $a$  es un residuo cuadrático en  $\mathbb{Z}_p$ , entonces  $a^{(p-1)/2} \pmod{p}$  es una raíz cuadrada de  $a$ , módulo  $p$ . ¿Cuánto tiempo se requiere para encontrar la raíz cuadrada de un residuo cuadrático  $a$  módulo  $p$ ?
- d. Describir un algoritmo aleatorio eficiente para encontrar un residuo no cuadrático, que no es un primo  $p$  arbitrario, es decir, un miembro del residuo  $\mathbb{Z}_p$  sea un residuo cuadrático. ¿Cuántas operaciones aritméticas requiere su algoritmo en promedio?

---

#### Notas del capítulo

Niven y Zuckerman [265] brindan una excelente introducción a la teoría elemental de los números. Knuth [210] contiene una buena discusión de algoritmos para encontrar el

máximo común divisor, así como otros algoritmos básicos de teoría de números. Bach [30] y Riesel [295] proporcionan estudios más recientes de la teoría numérica computacional. Dixon [91] ofrece una descripción general de las pruebas de factorización y primalidad. Las actas de congresos editadas por Pomerance [280] contienen varios artículos de encuesta excelentes. Más recientemente, Bach y Shallit [31] han proporcionado una visión general excepcional de los conceptos básicos de la teoría computacional de números.

Knuth [210] analiza el origen del algoritmo de Euclides. Aparece en el Libro 7, Proposiciones 1 y 2, de los Elementos del matemático griego Euclides, que fue escrito alrededor del 300 a. C. La descripción de Euclides puede haberse derivado de un algoritmo debido a Eudoxo alrededor del 375 a. C. El algoritmo de Euclides puede tener el honor de ser el más antiguo. algoritmo no trivial; sólo tiene rival en un algoritmo de multiplicación conocido por los antiguos egipcios. Shallit [312] relata la historia del análisis del algoritmo de Euclides.

Knuth atribuye un caso especial del teorema chino del resto (Theorem 31.27) al matemático chino Sun-Tsú, que vivió en algún momento entre el 200 a. C. y el 200 d. C.; la fecha es bastante incierta. El mismo caso especial fue presentado por el matemático griego Nicómaco alrededor del año 100 d. C. Fue generalizado por Chhin Chiu-Shao en 1247. El teorema chino del resto fue finalmente formulado y probado en toda su generalidad por L. Euler en 1734.

El algoritmo aleatorio de prueba de primalidad presentado aquí se debe a Miller [255] y Rabin [289]; es el algoritmo de prueba de primalidad aleatorio más rápido conocido, dentro de factores constantes. La demostración del Teorema 31.39 es una ligera adaptación de la sugerida por Bach [29]. Monier [258, 259] proporcionó una prueba de un resultado más fuerte para MILLER-RABIN . Durante muchos años, la prueba de primalidad fue el ejemplo clásico de un problema en el que la aleatorización parecía ser necesaria para obtener un algoritmo eficiente (de tiempo polinomial). En 2002, sin embargo, Agrawal, Kayal y Saxena [4] sorprendieron a todos con su algoritmo de prueba de primalidad de tiempo polinomial determinista. Hasta entonces, el algoritmo de prueba de primalidad determinista más rápido conocido, gracias a Cohen y Lenstra [73], se ejecutaba en el tiempo  $\lg n / O(\lg \lg \lg n)$  en la entrada  $n$ , que es ligeramente superpolinomial. No obstante, a efectos prácticos, los algoritmos aleatorios de prueba de primalidad siguen siendo más eficientes y son los preferidos.

El problema de encontrar grandes números primos "aleatorios" se analiza muy bien en un artículo por Beauchemin, Brassard, Crépeau, Goutier y Pomerance [36].

El concepto de criptosistema de clave pública se debe a Diffie y Hellman [87].

El criptosistema RSA fue propuesto en 1977 por Rivest, Shamir y Adleman [296]. Desde entonces, el campo de la criptografía ha florecido. Nuestra comprensión del sistema criptográfico RSA se ha profundizado y las implementaciones modernas utilizan refinamientos significativos de las técnicas básicas presentadas aquí. Además, se han desarrollado muchas técnicas nuevas para demostrar que los criptosistemas son seguros. Por ejemplo, Goldwasser y Micali [142] muestran que la aleatorización puede ser una herramienta eficaz en el diseño de esquemas seguros de cifrado de clave pública. Para esquemas de firma,

Goldwasser, Micali y Rivest [143] presentan un esquema de firma digital para el cual todo tipo de falsificación concebible es probablemente tan difícil como la factorización. Menezes, van Oorschot y Vanstone [254] brindan una descripción general de la criptografía aplicada.

Pollard [277] inventó la heurística rho para la factorización de enteros. La versión que aquí se presenta es una variante propuesta por Brent [56].

Los mejores algoritmos para factorizar números grandes tienen un tiempo de ejecución que crece aproximadamente exponencialmente con la raíz cúbica de la longitud del número  $n$  que se va a factorizar. El algoritmo general de factorización cribosa de campos numéricos (desarrollado por Bühler, Lenstra y Pomerance [57] como una extensión de las ideas del algoritmo de factorización cribosa de campos numéricos de Pollard [278] y Lenstra et al. [232] y refinado por Coppersmith [77] y otros) es quizás el algoritmo de este tipo más eficiente en general para grandes entradas. Aunque es difícil dar un análisis riguroso de este algoritmo, bajo suposiciones razonables podemos derivar una estimación del tiempo de ejecución de donde  $L_{1,2} \approx n^{1/3} \ln \ln n / L_{1,2} = 3; n^{1/3} \ln \ln n / 1.902 \ln 2$ ,

El método de la curva elíptica debido a Lenstra [233] puede ser más efectivo para algunas entradas que el método de criba de campos numéricos, ya que, al igual que el método rho de Pollard, puede encontrar un pequeño factor primo  $p$  con bastante rapidez. Con este método, el tiempo para encontrar  $p$  se estima en  $L_{1,2} \approx p^{1/2} \ln \ln p / 1.902 \ln 2$ .

## 32

## Coincidencia de cadenas

Los programas de edición de texto frecuentemente necesitan encontrar todas las apariciones de un patrón en el texto. Por lo general, el texto es un documento que se está editando y el patrón que se busca es una palabra particular proporcionada por el usuario. Los algoritmos eficientes para este problema, llamados "coincidencia de cadenas", pueden ayudar en gran medida a la capacidad de respuesta del programa de edición de texto. Entre sus muchas otras aplicaciones, los algoritmos de emparejamiento de cadenas buscan patrones particulares en las secuencias de ADN. Los motores de búsqueda de Internet también los utilizan para encontrar páginas web relevantes para las consultas.

Formalizamos el problema de coincidencia de cadenas de la siguiente manera. Suponemos que el texto es un arreglo  $T \in \Sigma^n$  de longitud  $n$  y que el patrón es un arreglo  $P \in \Sigma^m$  de longitud  $m$ . Suponemos además que los elementos de  $P$  y  $T$  son caracteres extraídos de un alfabeto finito  $\Sigma$ . Por ejemplo, podemos tener  $\Sigma = \{a, b, c, \dots, z\}$ . Las matrices de caracteres  $P$  y  $T$  a menudo se denominan cadenas de caracteres.

Haciendo referencia a la figura 32.1, decimos que el patrón  $P$  ocurre con el desplazamiento  $s$  en el texto  $T$  (o, de manera equivalente, que el patrón  $P$  ocurre comenzando en la posición  $s$  en el texto  $T$ ) si  $T[s:s+m-1] = P$  (es decir, si  $T[s:s+m-1] = P$ , entonces llamamos a un desplazamiento válido; de lo contrario, llamamos a  $s$  un desplazamiento no válido). El problema de coincidencia de cadenas es el problema de encontrar todos los desplazamientos válidos con los que un el patrón dado  $P$  ocurre en un texto dado  $T$ .

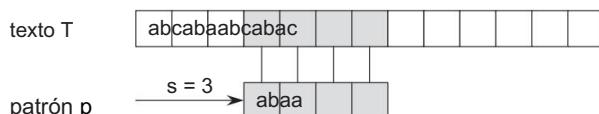


Figura 32.1 Un ejemplo del problema de coincidencia de cadenas, donde queremos encontrar todas las ocurrencias del patrón  $P$  en el texto  $T$ . El patrón ocurre solo una vez en el texto, en el turno  $s = 3$ , que llamamos un turno válido. Una línea vertical conecta cada carácter del patrón con su carácter coincidente en el texto y todos los caracteres coincidentes están sombreados.

Algoritmo	tiempo de preprocessamiento	Tiempo de
Ingenuo	0	coincidencia O..nm C
Rabin Karp	,m/	1/m/ O..nm C 1/m/ ,n/
autómata finito	Om j j/ ,m/	,n/
Knuth-Morris-Pratt		

Figura 32.2 Los algoritmos de emparejamiento de cadenas de este capítulo y sus tiempos de preprocessamiento y emparejamiento.

Excepto por el algoritmo ingenuo de fuerza bruta, que revisamos en la Sección 32.1, cada algoritmo de coincidencia de cadenas en este capítulo realiza algún procesamiento previo basado en el patrón y luego encuentra todos los cambios válidos; a esta última fase la llamamos “coincidencia”. La figura 32.2 muestra los tiempos de preprocessamiento y emparejamiento para cada uno de los algoritmos de este capítulo. El tiempo total de ejecución de cada algoritmo es la suma de los tiempos de preprocessamiento y emparejamiento. La Sección 32.2 presenta un interesante algoritmo de emparejamiento de cadenas, debido a Rabin y Karp. Aunque el tiempo de ejecución en el peor de los casos ,nm C 1/m/ de este algoritmo no es mejor que el del método ingenuo, funciona mucho mejor en promedio y en la práctica. También se generaliza muy bien a otros problemas de coincidencia de patrones. La sección 32.3 luego describe un algoritmo de emparejamiento de cadenas que comienza con la construcción de un autómata finito diseñado específicamente para buscar ocurrencias del patrón P dado en un texto. Este algoritmo toma Om j|j/ tiempo de preprocessamiento, pero solo ,n/ tiempo de coincidencia. La Sección 32.4 presenta el algoritmo Knuth-Morris-Pratt (o KMP) similar, pero mucho más inteligente; tiene el mismo tiempo de coincidencia ,n/ y reduce el tiempo de preprocessamiento a solo ,m/.

### Notación y terminología

Denotamos por  $\Sigma$  (léase “estrella sigma”) el conjunto de todas las cadenas de longitud finita formadas con caracteres del alfabeto  $\Sigma$ . En este capítulo, consideraremos solo cadenas de longitud finita. La cadena vacía de longitud cero, denotada "", también pertenece a  $\Sigma$ . La longitud de una cadena  $x$  se denota  $|x|$ . La concatenación de dos cadenas  $x$  e  $y$ , denotada  $xy$ , tiene una longitud  $|xy| = |x| + |y|$  y consta de los caracteres de  $x$  seguidos por los personajes de  $y$ .

Decimos que una cadena  $w$  es un prefijo de una cadena  $x$ , denotada  $w \leq x$ , si  $x \in D^*$  para alguna cadena  $y \in \Sigma^*$ . Tenga en cuenta que si  $w \leq x$ , entonces  $jwj \leq jxj$ . De manera similar, decimos que una cadena  $w$  es un sufijo de una cadena  $x$ , denotada  $w \geq x$ , si  $x \in D^*$  para alguna cadena  $y \in \Sigma^*$ . Al igual que con un prefijo,  $w \leq x$  implica  $jwj \leq jxj$ . Por ejemplo, tenemos  $ab \leq abcca$  y  $cca \leq abcca$ . La cadena vacía "" es tanto un sufijo como un prefijo de cada cadena. Para cualquier cadena  $x$  e  $y$  y cualquier carácter  $a$ , tenemos  $x \leq y$  si y solo si  $xa \leq ya$ .

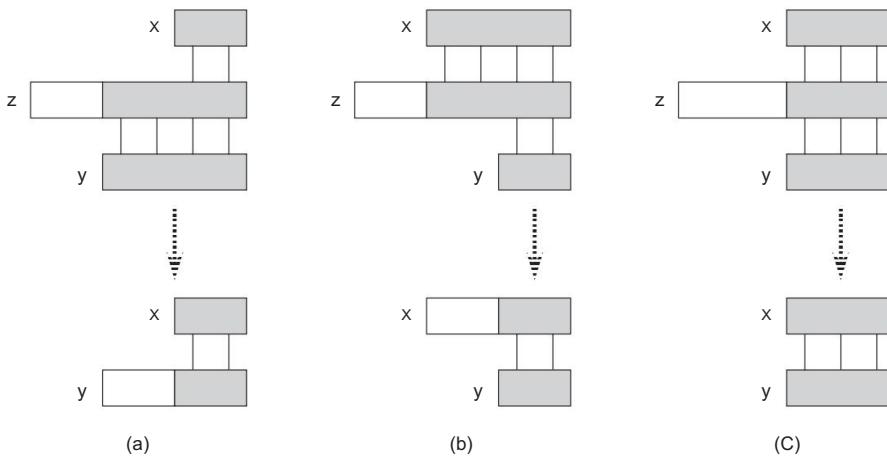


Figura 32.3 Una demostración gráfica del Lema 32.1. Supongamos que  $x \neq y$ . Las tres partes de la figura ilustran los tres casos del lema. Las líneas verticales conectan las regiones coincidentes (que se muestran sombreadas) de las cuerdas. (a) Si  $|x| < |y|$ , entonces  $x \neq y$ . (b) Si  $|x| > |y|$ , entonces  $y \neq x$ . (c) Si  $|x| = |y|$ , entonces  $x \neq y$ .

También tenga en cuenta que  $\leq$  y  $\geq$  son relaciones transitivas. El siguiente lema será útil más adelante.

### Lema 32.1 (Lema de sufijos superpuestos)

Supongamos que  $x$ ,  $y$  son cadenas tales que  $x \neq y$ . Si  $jxj$  entonces  $x = y$ . Si  $jxj > jyj$ ,  $jyj$ , entonces  $y \neq x$ . Si  $jxj \leq jyj$ , entonces  $x \neq D y$ .

Prueba Vea la Figura 32.3 para una prueba gráfica.

Para abreviar la notación, denotamos el prefijo de caracteres  $k$  P  $\infty 1 :: k$  del patrón P  $\infty 1 :: m$  por  $P_k$ . Por lo tanto,  $P_0 D''$  y  $P_m D P D P \infty 1 :: m$ . De manera similar, denotamos el prefijo de caracteres  $k$  del texto T por  $T_k$ . Usando esta notación, podemos establecer el problema de coincidencia de cadenas como el de encontrar todos los cambios s en el rango 0 snm tal que  $P \infty T_s C_m$ .

En nuestro pseudocódigo, permitimos que dos cadenas de igual longitud sean comparadas por igualdad como una operación primitiva. Si las cadenas se comparan de izquierda a derecha y la comparación se detiene cuando se descubre una discrepancia, suponemos que el tiempo que tarda dicha prueba es una función lineal del número de caracteres coincidentes descubiertos.

Para ser precisos, se supone que la prueba “ $x == y$ ” toma el tiempo  $\cdot t C 1/$ , donde  $t$  es la longitud de la cadena más larga ‘ tal que ‘  $x \neq y$  ‘  $y$ . (Escribimos  $\cdot t C 1/$  en lugar de  $\cdot t/$  para manejar el caso en el que  $t = 0$ ; los primeros caracteres comparados no coinciden, pero lleva una cantidad positiva de tiempo realizar esta comparación).

### 32.1 El algoritmo ingenuo de emparejamiento de cadenas

El algoritmo ingenuo encuentra todos los cambios válidos usando un ciclo que verifica la condición  $P \in C_1 : : m DT \in S C 1::s C m$  para cada uno de los  $n_m C 1$  valores posibles de  $s$ .

**NAIVE-STRING-MATCHER.T; P / 1 n D**

T:longitud 2 m D

P:longitud 3 para s D 0

a nm 4 si P  $\in C_1 : : m == T$

$\in S C 1::s C m$  imprime "Patrón ocurre con cambio" s

La figura 32.4 muestra el procedimiento ingenuo de emparejamiento de cadenas como si se deslizara una “plantilla” que contiene el patrón sobre el texto, notando que los cambios de todos los caracteres en la plantilla son iguales a los caracteres correspondientes en el texto. El ciclo for de las líneas 3 a 5 considera explícitamente cada cambio posible. La prueba en la línea 4 determina si el turno actual es válido; esta prueba se repite implícitamente para verificar las posiciones de los caracteres correspondientes hasta que todas las posiciones coincidan correctamente o se encuentre una falta de coincidencia. La línea 5 imprime cada turno válido.

El procedimiento NAIVE-STRING-MATCHER toma un tiempo  $O..nm C 1/m/$ , y este límite es ajustado en el peor de los casos. Por ejemplo, considere la cadena de texto an (una cadena de  $n_a$ ) y el patrón am. Para cada uno de los posibles valores  $n_m C 1$  del turno  $s$ , el ciclo implícito en la línea 4 para comparar los caracteres correspondientes debe ejecutarse  $m$  veces para validar el turno. El tiempo de ejecución del peor de los casos es, por lo tanto,  $..nm C 1/m/$ , que es  $.n2/$  si  $m D bn=2c$ . Debido a que no requiere preprocessamiento, el tiempo de ejecución de NAIVE STRING-MATCHER es igual al tiempo de coincidencia.

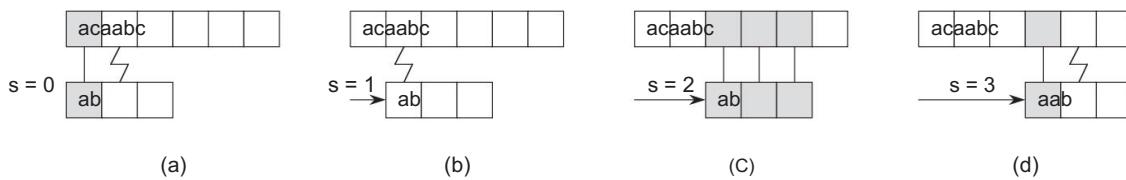


Figura 32.4 El funcionamiento del comparador de cadenas ingenuo para el patrón PD aab y el texto TD acaabc. Podemos imaginar el patrón P como una plantilla que deslizamos junto al texto. (a)-(d) Las cuatro alineaciones sucesivas probadas por el ingenuo comparador de cadenas. En cada parte, las líneas verticales conectan las regiones correspondientes que coinciden (se muestran sombreadas) y una línea irregular conecta el primer carácter que no coincide, si lo hay. El algoritmo encuentra una ocurrencia del patrón, en el desplazamiento s D 2, que se muestra en la parte (c).

Como veremos, NAIVE-STRING-MATCHER no es un procedimiento óptimo para este problema. De hecho, en este capítulo veremos que el algoritmo de Knuth-Morris-Pratt es mucho mejor en el peor de los casos. El emparejador de cadenas ingenuo es ineficiente porque ignora por completo la información obtenida sobre el texto para un valor de  $s$  cuando considera otros valores de  $s$ . Sin embargo, dicha información puede ser bastante valiosa. Por ejemplo, si  $PD \text{ aaab}$  y encontramos que  $s D 0$  es válido, entonces ninguno de los cambios 1, 2 o 3 son válidos, ya que  $T \not\in D b$ . En las siguientes secciones, examinamos varias maneras de hacer un uso efectivo de este tipo de información.

## Ejercicios

32.1-1

Muestre las comparaciones que hace el emparejador de cadenas ingenuas para el patrón PD 0001 en el texto TD 000010001010001.

32 1-2

Suponga que todos los caracteres en el patrón  $P$  son diferentes. Muestre cómo acelerar NAIVE-STRING-MATCHER para que se ejecute en el tiempo  $\Theta(n \cdot m)$  en un texto de  $n$  caracteres  $T$ .

32 1-3

Suponga que el patrón P y el texto T son cadenas elegidas al azar de longitud m y n, respectivamente, del alfabeto d-ario  $\{0, 1, \dots, d-1\}$ , donde d ≥ 2. Demuestre que el número esperado de comparaciones de carácter a carácter hechas por el ciclo implícito en la línea 4 del algoritmo ingenuo es

.nm C 1/1      d m      2.nm C 1/1

sobre todas las ejecuciones de este ciclo. (Suponga que el algoritmo ingenuo deja de comparar caracteres para un cambio dado una vez que encuentra una falta de coincidencia o coincide con el patrón completo). Por lo tanto, para cadenas elegidas al azar, el algoritmo ingenuo es bastante eficiente.

32 1-4

Supongamos que permitimos que el patrón P contenga ocurrencias de un carácter de espacio } que puede coincidir con una cadena arbitraria de caracteres (incluso uno de longitud cero). Por ejemplo, el patrón ab}ba\c{c aparece en el texto cabccbacbacab como

taxi							abdominalis
	'ba	"cba	'cc	'c		C	

y como

taxis      —ccbac'ba\_      'c      ab

Tenga en cuenta que el carácter de espacio puede aparecer un número arbitrario de veces en el patrón pero nunca en el texto. Proporcione un algoritmo de tiempo polinomial para determinar si tal patrón P ocurre en un texto T dado y analice el tiempo de ejecución de su algoritmo.

## 32.2 El algoritmo de Rabin-Karp

Rabin y Karp propusieron un algoritmo de coincidencia de cadenas que funciona bien en la práctica y que también se generaliza a otros algoritmos para problemas relacionados, como la coincidencia de patrones bidimensionales. El algoritmo de Rabin-Karp utiliza un tiempo de preprocesamiento  $,m$ , y su tiempo de ejecución en el peor de los casos es  $,nmC1/m$ . Sin embargo, según ciertas suposiciones, su tiempo de ejecución de caso promedio es mejor.

Este algoritmo hace uso de nociones elementales de teoría de números, como la equivalencia de dos números módulo un tercer número. Es posible que desee consultar la Sección 31.1 para las definiciones relevantes.

A efectos expositivos, supongamos que  $\dagger D f0; 1; 2; \dots; 9g$ , de modo que cada carácter sea un dígito decimal. (En el caso general, podemos suponer que cada carácter es un dígito en notación de base-d, donde  $d \leq j \leq j$ .) Entonces podemos ver una cadena de  $k$  caracteres consecutivos como si representara un número decimal de longitud- $k$ . La cadena de caracteres 31415 corresponde por tanto al número decimal 31.415. Debido a que interpretamos los caracteres de entrada como símbolos gráficos y dígitos, nos parece conveniente en esta sección indicarlos como lo haríamos con dígitos, en nuestra fuente de texto estándar.

Dado un patrón  $P \in \Sigma^k$ , sea  $p$  su valor decimal correspondiente. De manera similar, dado un texto  $T \in \Sigma^m$ , sea  $ts$  el valor decimal de la subcadena de longitud  $m$  en  $T$  de  $C_1 \dots C_m$ , para  $s \in D$ ;  $1; \dots; nm$ . Ciertamente,  $ts \equiv p \pmod{1}$  sólo si  $T[C_1 \dots C_m] = P$ ; por tanto,  $s$  es un desplazamiento válido si y sólo si  $ts \equiv p \pmod{1}$ . Si pudieramos calcular  $p$  en el tiempo  $,m$  y todos los valores de  $ts$  en un total de  $,nmC1$  tiempo,<sup>1</sup> entonces podríamos determinar todos los desplazamientos válidos  $s$  en el tiempo  $,m/n$  comparando  $p$  con cada uno de los valores de  $ts$ . (Por el momento, no nos preocupemos por la posibilidad de que los valores de  $p$  y  $ts$  sean números muy grandes).

Podemos calcular  $p$  en el tiempo  $,m$  usando la regla de Horner (ver Sección 30.1):

$$p = DP \cdot 10^m + DP \cdot 10^{m-1} \cdot C_1 + \dots + DP \cdot 10^1 \cdot C_{m-1} + C_m$$

De manera similar, podemos calcular  $t_0$  a partir de  $T[C_1 \dots C_m]$  en el tiempo  $,m$ .

<sup>1</sup>Escribimos  $,nmC1$  en lugar de  $,nm$  porque  $s$  toma en  $nm$  valores diferentes. El “C1” es significativo en un sentido asintótico porque cuando  $m \geq n$ , calcular el valor de  $ts$  toma  $,1$  tiempo, no  $,0$  tiempo.

Para calcular los valores restantes  $t_1; t_2; \dots; t_{nm}$  en el tiempo  $, nm/$ , observamos que podemos calcular  $tsC1$  a partir de  $ts$  en tiempo constante, ya que

$$tsC1 \leftarrow 10 \cdot ts + 10m1T \otimes C 1 / T \otimes C m C 1 \quad (32.1)$$

Restar  $10m1T \otimes C 1$  elimina el dígito de orden superior de  $ts$ , multiplicar el resultado por 10 desplaza el número a la izquierda en una posición de un dígito y sumar  $T \otimes C m C 1$  trae el dígito de orden inferior apropiado. Por ejemplo, si  $m = D 5$  y  $ts = D 31415$ , entonces deseamos eliminar el dígito de orden superior  $T \otimes C 1 D 3$  y traer el nuevo dígito de orden inferior (supongamos que es  $T \otimes C 5 C 1 D 2$ ) para obtener

$$\begin{aligned} tsC1 &\leftarrow 10 \cdot 31415 + 10000 / 3 / C 2 \\ &\leftarrow D14152 \end{aligned}$$

Si precalculamos la constante  $10m1$  (lo que podemos hacer en el tiempo  $O(\lg m)$  usando las técnicas de la Sección 31.6, aunque para esta aplicación es suficiente un método sencillo de tiempo  $O(m)$ ), entonces cada ejecución de la ecuación (32.1) toma una constante número de operaciones aritméticas. Por tanto, podemos calcular  $p$  en el tiempo  $, m/$ , y podemos calcular todo  $t_0; t_1; \dots; t_{nm}$  en el tiempo  $, nm C 1/$ . Por lo tanto, podemos encontrar todas las apariciones del patrón  $P \otimes C 1 : : m$  en el texto  $T \otimes C 1 : : n$  con  $, m/$  tiempo de preprocessamiento y  $, nm C 1/$  tiempo de coincidencia.

Hasta ahora, hemos pasado por alto intencionalmente un problema:  $p$  y  $ts$  pueden ser demasiado grandes para trabajar convenientemente. Si  $P$  contiene  $m$  caracteres, entonces no podemos suponer razonablemente que cada operación aritmética en  $p$  (que tiene  $m$  dígitos) toma "tiempo constante". Afortunadamente, podemos resolver este problema fácilmente, como muestra la figura 32.5: calcule  $p$  y los valores de  $ts$  módulo  $q$  adecuado. Podemos calcular  $p$  módulo  $q$  en  $, m/$  tiempo y todos los valores de  $ts$  módulo  $q$  en  $, nm C 1/$  tiempo. Si elegimos el módulo  $q$  como número primo, de modo que  $10q$  quepa en una palabra de computadora, entonces podemos realizar todos los cálculos necesarios con aritmética de precisión simple. En general, con un alfabeto de  $d$  caracteres  $f_0; f_1; \dots; f_{d-1}$ , elegimos  $q$  para que  $dq$  quepa dentro de una palabra de computadora y ajustamos la ecuación de recurrencia (32.1) para trabajar módulo  $q$ , de modo que se convierte en

$$tsC1 \leftarrow .d \cdot ts + T \otimes C 1h / CT \otimes C m C 1 / \text{mod } q ; \quad (32.2)$$

donde  $hd .mod q^1$  es el valor del dígito "1" en la posición de orden superior de una ventana de texto de  $m$  dígitos.

Sin embargo, la solución de trabajar módulo  $q$  no es perfecta:  $ts \equiv p \pmod{q}$  no implica que  $ts = p$ . Por otro lado, si  $ts \neq p \pmod{q}$ , entonces definitivamente tenemos que  $ts \neq p$ , por lo que shift  $s$  no es válido. Por lo tanto, podemos usar la prueba  $ts \equiv p \pmod{q}$  como una prueba heurística rápida para descartar cambios no válidos en  $s$ . Cualquier cambio en  $s$  para el cual  $ts \equiv p \pmod{q}$  debe probarse más para ver si  $s$  es realmente válido o si solo tenemos un resultado falso. Esta prueba adicional verifica explícitamente la condición

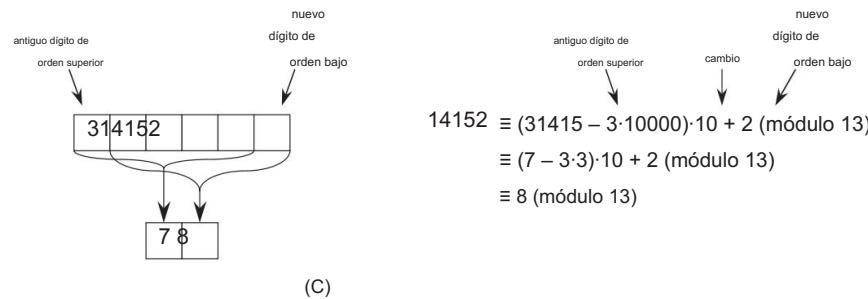
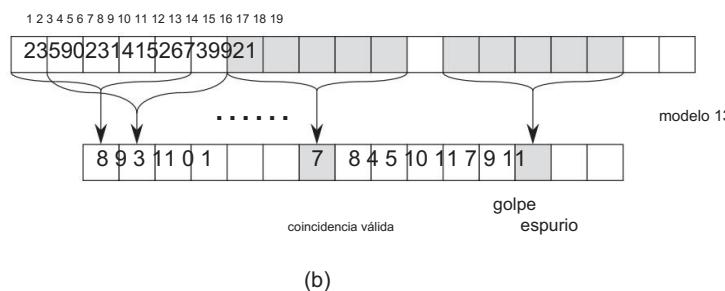
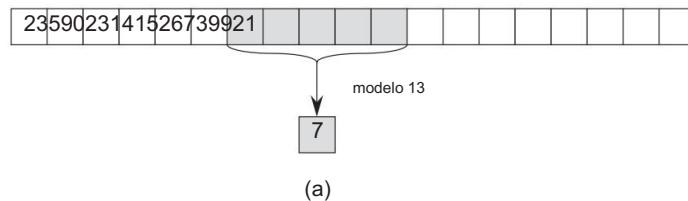


Figura 32.5 El algoritmo de Rabin-Karp. Cada carácter es un dígito decimal y calculamos valores módulo 13. (a) Una cadena de texto. Una ventana de longitud 5 se muestra sombreada. El valor numérico del número sombreado, calculado en módulo 13, da el valor 7. (b) La misma cadena de texto con valores calculados en módulo 13 para cada posición posible de una ventana de longitud 5. Asumiendo que el patrón PD 31415, buscamos ventanas cuyo valor módulo 13 sea 7, ya que  $31415 \equiv 7 \pmod{13}$ . El algoritmo encuentra dos ventanas de este tipo, que se muestran sombreadas en la figura. El primero, que comienza en la posición de texto 7, es de hecho una ocurrencia del patrón, mientras que el segundo, que comienza en la posición de texto 13, es un acierto falso. (c) Cómo calcular el valor de una ventana en tiempo constante, dado el valor de la ventana anterior. La primera ventana tiene el valor 31415. Si se elimina el dígito de orden superior 3, se desplaza a la izquierda (multiplicando por 10) y luego se suma el dígito de orden inferior 2, se obtiene el nuevo valor 14152. Debido a que todos los cálculos se realizan en módulo 13, el valor para la primera ventana es 7 y el valor para la nueva ventana es 8.

$P \in \{0, 1, \dots, m-1\}^m$  es el patrón,  $C \in \{0, 1, \dots, m-1\}^n$  es la cadena. Si  $q$  es lo suficientemente grande, entonces esperamos que los aciertos espurios ocurrían con la suficiente poca frecuencia como para que el costo de la verificación adicional sea bajo.

El siguiente procedimiento hace que estas ideas sean precisas. Las entradas al procedimiento son el texto  $T$ , el patrón  $P$ , la raíz  $d$  a usar (que normalmente se toma como  $j+1$ ), y el primo  $q$  usar.

```
RABIN-KARP-MATCHER(T; PAG; d; q/ 1 n D
T:longitud 2 m D
P:longitud mod q
3 h D d 4 p m1
D 0 5 t0 D 0
6 para i D 1
a mp D .dp CP OEi/ // preprocesamiento
7 mod q t0 D .dt0 CT OEi/ mod q

8 9 para s D 0 a n m si p == // coincidencia
10 ts si P
11 OE1 : : m == T OEs C 1::s C m imprime
12 "Patrón ocurre con cambio" s si s<n m
13
14 tsC1 D .d.ts T OEs C 1h/ CT OEs C m C 1/ mod q
```

El procedimiento RABIN-KARP-MATCHER funciona de la siguiente manera. Todos los caracteres se interpretan como dígitos radix- $d$ . Los subíndices en  $t$  se proporcionan solo para mayor claridad; el programa funciona correctamente si se eliminan todos los subíndices. La línea 3 inicializa  $h$  al valor de la posición del dígito de orden superior de una ventana de dígitos  $m$ . Las líneas 4 a 8 calculan  $p$  como el valor de  $P \in \{0, 1, \dots, m-1\}^m$  y  $t_0$  como el valor de  $T \in \{0, 1, \dots, m-1\}^n$ . El ciclo for de las líneas 9 a 14 itera a través de todos los cambios posibles, manteniendo el siguiente invariante:

Cada vez que se ejecuta la línea 10,  $ts \in \{0, 1, \dots, m-1\}^m$ .

Si  $p \in D$  en la línea 10 (un “acuerdo”), entonces la línea 11 verifica si  $P \in \{0, 1, \dots, m-1\}^m$  para descartar la posibilidad de un acuerdo falso. La línea 12 imprime cualquier turno válido que se encuentre. Si  $s < n$  (marcado en la línea 13), entonces el bucle for se ejecutará al menos una vez más, por lo que la línea 14 se ejecuta primero para garantizar que el bucle se mantiene invariable cuando volvemos a la línea 10. La línea 14 calcula el valor de  $tsC_1 \mod q$  del valor de  $ts \mod q$  en tiempo constante usando la ecuación (32.2) directamente.

RABIN-KARP-MATCHER toma un tiempo de preprocesamiento  $\sim m^n$ , y su tiempo de coincidencia es  $\sim nm C 1/m$  en el peor de los casos, ya que (al igual que el algoritmo ingenuo de coincidencia de cadenas) el algoritmo Rabin-Karp verifica explícitamente cada turno válido. Si soy PD

y TD an, entonces la verificación lleva un tiempo  $\dots nmC1/m$ , ya que cada uno de los posibles desplazamientos de nmC1 es válido.

En muchas aplicaciones, esperamos pocos cambios válidos, tal vez alguna constante c de ellos. En tales aplicaciones, el tiempo de coincidencia esperado del algoritmo es solo  $O(nmC1/Ccm/DOnCm)$ , más el tiempo requerido para procesar coincidencias espurias. Podemos basar un análisis heurístico en la suposición de que la reducción de valores módulo q actúa como un mapeo aleatorio de  $\mathbb{T}$  a  $Zq$ . (Consulte la discusión sobre el uso de la división para el hashing en la Sección 11.3.1. Es difícil formalizar y probar tal suposición, aunque un enfoque viable es suponer que q se elige al azar entre números enteros del tamaño apropiado. No lo haremos seguir esta formalización aquí.)

Entonces podemos esperar que el número de aciertos espurios sea  $On=q/$ , ya que podemos estimar la probabilidad de que un ts arbitrario sea equivalente a p, módulo q, como  $1=q$ .

Dado que hay posiciones On/ en las que falla la prueba de la línea 10 y gastamos Om/ tiempo para cada acierto, el tiempo de coincidencia esperado que toma el algoritmo de Rabin-Karp es

$On/C Om C n=q//$  ;

donde es el número de turnos válidos. Este tiempo de ejecución es  $On/si D O.1/$  y elegimos q m. Es decir, si el número esperado de cambios válidos es pequeño ( $O.1/$ ) y elegimos que el primo q sea mayor que la longitud del patrón, entonces podemos esperar que el procedimiento de Rabin-Karp use solo  $On C m/$  tiempo coincidente. Desde mn, este tiempo esperado de coincidencia es  $On/$ .

## Ejercicios

### 32.2-1

Trabajando módulo q D 11, ¿cuántos aciertos falsos detecta el comparador Rabin-Karp en el texto TD 3141592653589793 al buscar el patrón PD 26?

### 32.2-2

¿Cómo extendería el método de Rabin-Karp al problema de buscar en una cadena de texto una ocurrencia de cualquiera de un conjunto dado de k patrones? Comience asumiendo que todos los k patrones tienen la misma longitud. Luego, generaliza tu solución para permitir que los patrones tengan diferentes longitudes.

### 32.2-3

Muestre cómo extender el método de Rabin-Karp para manejar el problema de buscar un patrón de mm dado en una matriz de caracteres nn. (El patrón se puede desplazar vertical y horizontalmente, pero no se puede girar).

## 32.2-4

Alice tiene una copia de un archivo largo de  $n$  bits  $AD$  han<sub>1</sub>; an<sub>2</sub>;...;a<sub>0i</sub>, y Bob también tiene un archivo de  $n$  bits  $BD$  hbn<sub>1</sub>; bn<sub>2</sub>;...;b<sub>0i</sub>. Alice y Bob desean saber si sus archivos son idénticos. Para evitar transmitir todo A o B, utilizan la siguiente comprobación probabilística rápida. Juntos, seleccionan un número primo  $q > 1000n$  y seleccionan al azar un número entero  $x$  de  $f_0; 1; \dots; q - 1$ . Luego, Alicia evalúa

$Ax/D Xn_1$  aixi !  
iD0 modo q

y Bob evalúa de manera similar  $Bx/$ . Demuestre que si  $A \neq B$ , hay como máximo una probabilidad entre 1000 de que  $Ax/D Bx/$ , mientras que si los dos archivos son iguales,  $Ax/$  es necesariamente el mismo que  $Bx/$ . (Sugerencia: vea el ejercicio 31.4-4.)

## 32.3 Coincidencia de cadenas con autómatas finitos

Muchos algoritmos de emparejamiento de cadenas construyen un autómata finito, una máquina simple para procesar información, que escanea la cadena de texto T en busca de todas las ocurrencias del patrón P. Esta sección presenta un método para construir dicho autómata. Estos autómatas de coincidencia de cadenas son muy eficientes: examinan cada carácter de texto exactamente una vez, tomando un tiempo constante por carácter de texto. El tiempo de coincidencia utilizado, después de preprocesar el patrón para construir el autómata, es por lo tanto  $\sim n$ . Sin embargo, el tiempo para construir el autómata puede ser grande si  $\tau$  es grande. La Sección 32.4 describe una forma inteligente de solucionar este problema.

Comenzamos esta sección con la definición de un autómata finito. Luego examinamos un autómata especial de coincidencia de cadenas y mostramos cómo usarlo para encontrar ocurrencias de un patrón en un texto. Finalmente, mostraremos cómo construir el autómata de coincidencia de cadenas para un patrón de entrada dado.

## autómatas finitos

Un autómata finito M, ilustrado en la figura 32.6, es una tupla de  $5 . Q; q_0; A; \tau; \delta$ ; donde yo

Q es un conjunto finito de estados,

$q_0 \in Q$  es el estado inicial,

AQ es un conjunto distinguido de estados de aceptación,

$\tau$  es un alfabeto de entrada finito,

$\delta$  es una función de  $Q \times \tau$  en  $Q$ , llamada función de transición de M.

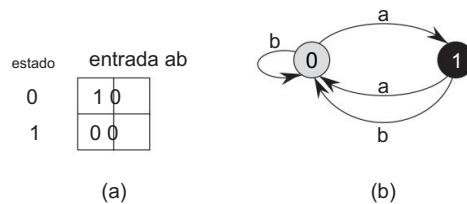


Figura 32.6 Un autómata finito simple de dos estados con un conjunto de estados QD f0; 1g, estado inicial q0 D 0 y alfabeto de entrada  $\{\text{f}, \text{d}\}$ ;  $\{\text{a}, \text{b}\}$ . (a) Una representación tabular de la función de transición  $\delta$ . (b) Un diagrama de transición de estado equivalente. El estado 1, que se muestra en negro, es el único estado de aceptación. Los bordes dirigidos representan transiciones. Por ejemplo, el borde del estado 1 al estado 0 etiquetado como b indica que  $\delta(1, b) = 0$ . Este autómata acepta aquellas cadenas que terminan en un número impar de aes. Más precisamente, acepta una cadena  $x$  si y solo si  $x \in D^*$ , donde  $D = \{a\} \cup \{b\}^k$  y termina con  $ab$ , y  $k$  es impar. Por ejemplo, en la entrada abaaa, incluido el estado inicial, este autómata entra en la secuencia de estados h0; 1; 0; 1; 0; 1, por lo que acepta esta entrada. Para la entrada abbaa, entra en la secuencia de estados h0; 1; 0; 0; 1; 0i, por lo que rechaza esta entrada.

El autómata finito comienza en el estado  $q_0$  y lee los caracteres de su cadena de entrada uno a la vez. Si el autómata está en el estado  $q$  y lee el carácter de entrada  $a$ , se mueve (“hace una transición”) del estado  $q$  al estado  $\delta(q; a)$ . Siempre que su estado actual  $q$  sea miembro de  $A$ , la máquina  $M$  ha aceptado la cadena leída hasta el momento. Una entrada que no se acepta se rechaza.

Un autómata finito  $M$  induce una función  $\delta_M : \Sigma^* \rightarrow A$ , llamada función de estado final, de  $\Sigma^*$  a  $A$  tal que  $.w/\epsilon$  es el estado en el que  $M$  termina después de escanear la cadena  $w$ . Así,  $M$  acepta una cadena  $w$  si y sólo si  $.w/\epsilon \in A$ . Definimos la función recursivamente, usando la función de transición:

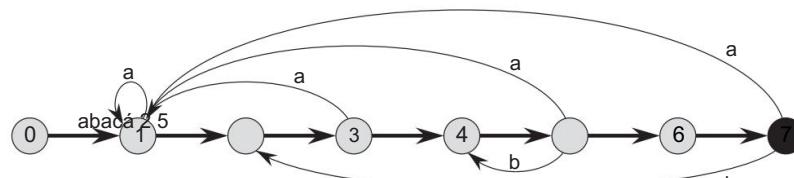
."/ D q0 ; .wa/

D l..w/; a/ para w 2 t; a 2 t

Autómatas de coincidencia de cadenas

Para un patrón  $P$  dado, construimos un autómata de coincidencia de cadenas en un paso de preprocessamiento antes de usarlo para buscar la cadena de texto. La figura 32.7 ilustra cómo construimos el autómata para el patrón  $PD$  ababaca. De ahora en adelante, supondremos que  $P$  es una cadena de patrón fijo dado; por brevedad, no indicaremos la dependencia de  $P$  en nuestra notación.

Para especificar el autómata de emparejamiento de cadenas correspondiente a un patrón dado  $P \in \Sigma^*$ , primero definimos una función auxiliar llamada función de sufijo correspondiente a  $P$ . La función asigna  $\dagger$  a  $f_0; 1; \dots; m$  tal que  $x.$  es la longitud del prefijo más largo de  $P$  que también es un sufijo de  $x$ :



(a)

estado	entrada abc	PAG
0	1 0 0	a
1	1 2 0	b
2	3 0 0	a
3	1 4 0	b
4	5 0 0	a
5	1 4 6	c
6	7 0 0	T
7	1 2 0	OEi estado .Ti/

(b)

i — 1 2 3 4 5 6 7 8 9 10 11  
T — abababacaba  
OEi estado .Ti/ 012345456 2 3

(C)

Figura 32.7 (a) Un diagrama de transición de estado para el autómata de comparación de cadenas que acepta todas las cadenas que terminan en la cadena ababaca. El estado 0 es el estado inicial y el estado 7 (ennegrecido) es el único estado de aceptación. Un borde dirigido del estado i al estado j etiquetado como a/j i; a/Dj. Los bordes hacia la derecha que forman la "columna vertebral" del autómata, que se muestran gruesos en la figura, corresponden a coincidencias exitosas entre el patrón y los caracteres de entrada. Los bordes hacia la izquierda corresponden a coincidencias fallidas. Se omiten algunos bordes correspondientes a coincidencias fallidas; por convención, si un estado i no tiene borde saliente etiquetado como a para algún a, entonces a/i; a/D0. (b) La función de transición correspondiente i, y la cadena patrón PD ababaca. Las entradas correspondientes a coincidencias exitosas entre el patrón y los caracteres de entrada se muestran sombreadas. (c) El funcionamiento del autómata sobre el texto TD abababacaba. Debajo de cada carácter de texto T aparece el estado .Ti/ en el que se encuentra el autómata después de procesar el prefijo Ti. El autómata encuentra una ocurrencia del patrón, que termina en la posición 9.

La función de sufijo está bien definida ya que la cadena vacía P0 D " es un sufijo de cada cadena. Como ejemplos, para el patrón PD ab, tenemos ./D 0, .ccaca/ D 1 y .ccab/ D 2 Para un patrón P de longitud m, tenemos .x/ D m si y solo si P = x. De la definición de la función sufijo, x = y implica .x/ = .y/.

Definimos el autómata de coincidencia de cadenas que corresponde a un patrón dado P OE1 : : m como sigue:

El conjunto de estados Q es  $f_0; 1; \dots; m$ . El estado inicial  $q_0$  es el estado 0 y el estado  $m$  es el único estado de aceptación.

La función de transición  $\delta$  está definida por la siguiente ecuación, para cualquier estado  $q$  y carácter  $a$ :

$$\delta(q; a) = D.P_{qa} : \quad (32.4)$$

Definimos  $\delta(q; a) = D.P_{qa}$  porque queremos realizar un seguimiento del prefijo más largo del patrón  $P$  que ha coincidido con la cadena de texto  $T$  hasta el momento. Consideramos los caracteres leídos más recientemente de  $T$ . Para que una subcadena de  $T$  —digamos la subcadena que termina en  $T[i]$ — coincida con algún prefijo  $P_j$  de  $P$ , este prefijo  $P_j$  debe ser un sufijo de  $T[i]$ . Supongamos que  $q \in D.T[i]$ , de modo que después de leer  $a$  el autómata está en  $T[i+1]$ , enunciando  $q$ . Diseñamos la función de transición  $\delta$  para que este número de estado,  $q$ , nos diga la longitud del prefijo más largo de  $P$  que coincide con un sufijo de  $T[i]$ . Es decir, en el estado  $q$ ,  $P_q = T[i]q \in D.T[i]$ . (Siempre que  $q \in D$ , todos los  $m$  caracteres de  $P$  coinciden con un sufijo de  $T[i]$ , por lo que hemos encontrado una coincidencia.) Por lo tanto, dado que  $D.T[i]$  y  $D.T[i+1]$  son iguales a  $q$ , veremos (en el Teorema 32.4, a continuación) que el autómata mantiene el siguiente invariante:

$$D.T[i] = D.T[i+1] : \quad (32.5)$$

Si el autómata está en el estado  $q$  y lee el siguiente carácter  $T[i+1] \in C$ , entonces queremos que la transición conduzca al estado correspondiente al prefijo más largo de  $P$  que es un sufijo de  $T[i]$ , y ese estado es  $D.T[i]$ . Debido a que  $P_q$  es el prefijo más largo de  $P$  que es un sufijo de  $T[i]$ , el prefijo más largo de  $P$  que es un sufijo de  $T[i]$  no solo es  $D.T[i]$ , sino también  $D.P_q$ . (El Lema 32.3, en la página 1000, prueba que  $D.T[i] = D.P_q$ .)

Así, cuando el autómata está en el estado  $q$ , queremos que la función de transición sobre el carácter  $a$  lleve al autómata al estado  $D.P_q$ .

Hay dos casos a considerar. En el primer caso, si  $a \in P$  ( $C \cap P \neq \emptyset$ ), de modo que el carácter  $a$  sigue coincidiendo con el patrón; en este caso, porque  $\delta(q; a) = q$ , la transición continúa a lo largo de la “espina dorsal” del autómata (los bordes gruesos en la Figura 32.7). En el segundo caso, si  $a \notin P$  ( $C \cap P = \emptyset$ ), por lo que  $a$  no sigue el patrón. Aquí, debemos encontrar un prefijo más pequeño de  $P$  que también sea un sufijo de  $T[i]$ .

Debido a que el paso de preprocessamiento compara el patrón consigo mismo al crear el autómata de coincidencia de cadenas, la función de transición identifica rápidamente el prefijo más largo de  $P$ .

Veamos un ejemplo. El autómata de emparejamiento de cadenas de la figura 32.7 tiene 1.5; c/ D 6, ilustrando el primer caso, en el que continúa el partido. Para ilustrar el segundo caso, observe que el autómata de la figura 32.7 tiene 1.5; b/D 4.

Hacemos esta transición porque si el autómata lee ab en el estado q D 5, entonces Pqb D ababab, y el prefijo más largo de P que también es un sufijo de ababab es P4 D abab.

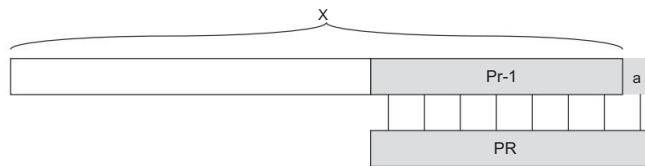


Figura 32.8 Una ilustración para la prueba del Lema 32.2. La figura muestra que  $r \in D .xa/$ .  $.x/ \in C 1,$

Para aclarar la operación de un autómata de coincidencia de cadenas, ahora proporcionamos un programa simple y eficiente para simular el comportamiento de dicho autómata (representado por su función de transición  $\tau$ ) al encontrar ocurrencias de un patrón  $P$  de longitud  $m$  en un texto de entrada  $T \in \Sigma^*$ . Como para cualquier autómata de comparación de cadenas para un patrón de longitud  $m$ , el conjunto de estados  $Q$  es  $f_0; f_1; \dots; f_m$ , el estado de inicio es  $f_0$  y el único estado de aceptación es el estado  $f_m$ .

**FINITO-AUTOMATON-MATCHER.**  $T; i; m / 1 \in D T:$

longitud 2  $q \in D 0 3$

para  $i \in D 1 a$

$n \in D 4 5 6$

$q \in D \tau(q; T[i:i+m]) \text{ si } q$

$\equiv m$

imprime "El patrón ocurre con el cambio" im

A partir de la estructura de bucle simple de FINITE-AUTOMATON-MATCHER, podemos ver fácilmente que su tiempo de coincidencia en una cadena de texto de longitud  $n$  es  $.n/$ . Sin embargo, este tiempo de coincidencia no incluye el tiempo de preprocessamiento requerido para calcular la función de transición  $\tau$ . Abordaremos este problema más adelante, después de probar primero que el procedimiento FINITE-AUTOMATON-MATCHER funciona correctamente.

Considere cómo opera el autómata en un texto de entrada  $T \in \Sigma^*$ . Probaremos que el autómata está en el estado  $.T[i]$  después de escanear el carácter  $T[i]$ . Dado que  $.T[i] \in D m$  si y solo si  $P = T[i : i+m]$ , la máquina está en el estado de aceptación  $m$  si y solo si acaba de escanear el patrón  $P$ . Para probar este resultado, hacemos uso de los siguientes dos lemas sobre el sufijo función  $.$

#### Lema 32.2 (Desigualdad sufijo-función)

Para cualquier cadena  $x$  y carácter  $a$ , tenemos  $.xa/$

$.x/ \in C 1.$

Prueba Haciendo referencia a la figura 32.8, sea  $r \in D .xa/$ . Si  $r \in D 0$ , entonces la conclusión  $.xa/ \in D .x/$  se cumple trivialmente, por la no negatividad de  $.x/$ . Ahora suponga que  $r > 0$ . Entonces,  $Pr = xa$ , por la definición de  $.$  Así,  $Pr \in x$ , por

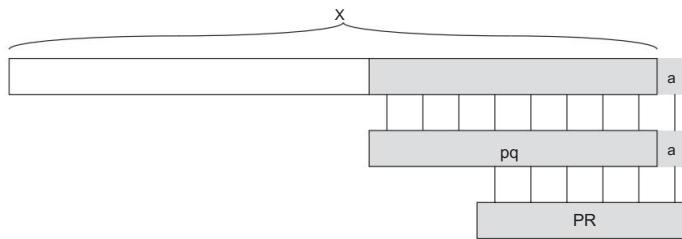


Figura 32.9 Una ilustración para la prueba del Lema 32.3. La figura muestra que  $r \in D .Pqa/$ , donde  $q \in D .x/$  y  $r \in D .xa/$ .

quitando la  $a$  del final de  $Pr$  y del final de  $xa$ . Por lo tanto,  $r \in D .x/$ , ya que  $.x/$  es el mayor  $k$  tal que  $P_k \in x$ , y por lo tanto  $.xa/ \in D .r .x/C 1. \blacksquare$

#### Lema 32.3 (Lema de recursividad sufijo-función)

Para cualquier cadena  $x$  y carácter  $a$ , si  $q \in D .x/$ , entonces  $.xa/ \in D .Pqa/$ .

Prueba De la definición de tenemos  $Pq \in x$ . Como muestra la figura 32.9, también tenemos  $Pqa \in xa$ . Si hacemos  $r \in D .xa/$ , entonces  $Pr \in xa$  y, por el Lema 32.2,  $r \in C 1$ . Así, tenemos  $jPr \in D .rq \in C 1 \in D .jPqaj$ . Como  $Pqa \in xa$ ,  $Pr \in xa$  y  $jPr \in jPqaj$ , el Lema 32.1 implica que  $Pr \in Pqa$ . Por tanto,  $r \in Pqa/$ , es decir,  $.xa/ \in Pqa/$ . Pero también tenemos  $.Pqa/ \in .xa/$ , ya que  $Pqa \in xa$ .

Así,  $.xa/ \in D .Pqa/$ .  $\blacksquare$

Ahora estamos listos para demostrar nuestro teorema principal que caracteriza el comportamiento de un autómata de coincidencia de cadenas en un texto de entrada dado. Como se señaló anteriormente, este teorema muestra que el autómata simplemente lleva la cuenta, en cada paso, del prefijo más largo del patrón que es un sufijo de lo que se ha leído hasta ahora. En otras palabras, el autómata mantiene la invariante (32.5).

#### Teorema 32.4 Si

es la función de estado final de un autómata de coincidencia de cadenas para un patrón dado  $P$  y  $T \in \Sigma^*$  es un texto de entrada para el autómata, entonces

$$\cdot T_i / \in D .T_i /$$

para  $i \in \{0, 1, \dots\}$  norte.

Prueba La prueba es por inducción sobre  $i$ . Para  $i = 0$ , el teorema es trivialmente verdadero, ya que  $T_0 \in D ". Por lo tanto,  $.T_0 / \in D .T_0 /$ .$

Ahora, asumimos que  $.Ti \in D .Ti$  y demostramos que  $.TiC1 \in D .TiC1$ . Sea  $q$  denotar  $.Ti$ , y sea  $a \in C$ . Entonces,

$.TiC1 \in D .Tia \in D .Ti$ ; (por las definiciones de  $TiC1$  y  $a$ )  
 $a \in (por la definición de )$   
 $D .i.q; a \in (por la definición de q)$   
 $D .Pqa \in (por la definición (32.4) de i)$   
 $D .Tia \in (por Lema 32.3 e inducción)$   
 $D .TiC1 \in (según la definición de  $TiC1$ )$ . ■

Por el teorema 32.4, si la máquina entra en el estado  $q$  en la línea 4, entonces  $q$  es el mayor valor tal que  $Pq \in Ti$ . Por lo tanto, tenemos  $q \in D$  en la línea 5 si y sólo si la máquina acaba de escanear una ocurrencia del patrón  $P$ . Concluimos que FINITE AUTOMATON-MATCHER opera correctamente.

#### Cálculo de la función de transición

El siguiente procedimiento calcula la función de transición  $i$  a partir de un patrón dado  $P$  1 : : metro.

#### CÁLCULO-TRANSICIÓN-FUNCIÓN.P; $\dagger / 1 m D P$ :

```

longitud 2 para q D 0
a m 3 para cada
carácter a 2 †
4      k D min.m C 1; q C 2/ repetir
5      k D k 1
6
7      hasta Pk = Pqa
           i.q; a/ D k 8 9
volver i

```

Este procedimiento calcula  $i.q; a/$  de manera directa de acuerdo con su definición en la ecuación (32.4). Los bucles anidados que comienzan en las líneas 2 y 3 consideran todos los estados  $q$  y todos los caracteres  $a$ , y las líneas 4–8 establecen  $i.q; a/$  sea el mayor  $k$  tal que  $Pk = Pqa$ . El código comienza con el mayor valor concebible de  $k$ , que es  $\min.m; q C 1/$ . Luego decrece  $k$  hasta  $Pk = Pqa$ , lo que eventualmente debe ocurrir, ya que  $P0 D "$  es un sufijo de cada cadena.

El tiempo de ejecución de COMPUTE-TRANSITION-FUNCTION es  $O.m3 j†j/$ , porque los bucles externos contribuyen con un factor de  $m j†j$ , el bucle de repetición interno puede ejecutarse como máximo  $m C 1$  veces, y la prueba  $Pk = Pqa$  en la línea 7 puede requerir una comparación

a m caracteres. Existen procedimientos mucho más rápidos; al utilizar alguna información hábilmente calculada sobre el patrón P (vea el ejercicio 32.4-8), podemos mejorar el tiempo requerido para calcular  $i$  de  $P$  a  $\Omega_m jTj$ . Con este procedimiento mejorado para calcular  $i$ , podemos encontrar todas las apariciones de un patrón de longitud  $m$  en un texto de longitud  $n$  sobre un alfabeto  $\Sigma$  con  $\Omega_m jTj$  tiempo de preprocesamiento y  $,n/m$  tiempo de coincidencia.

### Ejercicios

#### 32.3-1

Construya el autómata de coincidencia de cadenas para el patrón PD aabab e ilustre su operación en la cadena de texto TD aaababaabaababaab.

#### 32.3-2

Dibuje un diagrama de transición de estado para un autómata de combinación de cadenas para el patrón ababbabbabbabbabb sobre el alfabeto  $\Sigma = \{a, b\}$ .

#### 32.3-3

Llamamos a un patrón  $P$  no superpuesto si  $P_k = P_q$  implica  $k = q$ . Describa el diagrama de transición de estado del autómata de coincidencia de cadenas para un patrón que no se superpone.

#### 32.3-4 ?

Dados dos patrones,  $P$  y  $P_0$  , Describir cómo construir un autómata finito que determinan todas las ocurrencias de cualquier patrón. Intente minimizar el número de estados en su autómata.

#### 32.3-5

Dado un patrón  $P$  que contiene caracteres en blanco (vea el ejercicio 32.1-4), muestre cómo construir un autómata finito que pueda encontrar una ocurrencia de  $P$  en un texto  $T$  en el tiempo de encendido/coincidencia, donde  $n = |T|$ .

## ? 32.4 El algoritmo de Knuth-Morris-Pratt

Ahora presentamos un algoritmo de coincidencia de cadenas de tiempo lineal debido a Knuth, Morris y Pratt. Este algoritmo evita calcular la función de transición  $i$  por completo, y su tiempo de coincidencia es  $,m/n$  usando solo una función auxiliar que calculamos previamente a partir del patrón en el tiempo  $,m$  y almacenamos en una matriz  $\Omega_{m+1} \times m$ . La matriz nos permite calcular la función de transición  $i$  de manera eficiente (en un sentido amortizado) "sobre la marcha" según sea necesario. En términos generales, para cualquier estado  $q \in Q$ ;  $1 \leq i \leq m$  y cualquier carácter

a  $2^{\lfloor \frac{m}{k} \rfloor}$ , el valor  $\text{CEq}$  contiene la información que necesitamos para calcular  $i.q$ ;  $a/q$  pero eso no depende de  $a$ . Dado que la matriz tiene solo  $m$  entradas, mientras que  $i$  tiene  $\lfloor \frac{m}{k} \rfloor$  entradas, ahorraremos un factor de  $j \lfloor \frac{m}{k} \rfloor$  en el tiempo de preprocesamiento al calcular en lugar de  $i$ .

La función de prefijo para un patrón.

La función de prefijo para un patrón encapsula el conocimiento sobre cómo el patrón coincide con los cambios de sí mismo. Podemos aprovechar esta información para evitar probar cambios inútiles en el ingenuo algoritmo de coincidencia de patrones y evitar calcular previamente la función de transición completa  $i$  para un autómata de coincidencia de cadenas.

Considere la operación del comparador de cadenas ingenuo. La figura 32.10(a) muestra un cambio particular  $s$  de una plantilla que contiene el patrón PD ababaca contra un texto T . Para este ejemplo, q D 5 de los caracteres se han emparejado correctamente, pero el sexto carácter del patrón no logra coincidir con el carácter de texto correspondiente. La información de q caracteres han coincidido con éxito determina los caracteres de texto correspondientes. Conocer estos caracteres de texto q nos permite determinar de inmediato que ciertos turnos no son válidos. En el ejemplo de la figura, el desplazamiento  $s C 1$  es necesariamente inválido, ya que el primer carácter del patrón (a) estaría alineado con un carácter de texto que sabemos que no coincide con el primer carácter del patrón, pero sí con el segundo carácter del patrón ( b). El cambio  $s0 D s C 2$  que se muestra en la parte (b) de la figura, sin embargo, alinea los primeros tres caracteres del patrón con tres caracteres de texto que necesariamente deben coincidir. En general, es útil saber la respuesta a la siguiente pregunta:

Dado que los caracteres de patrón  $P \text{CE1} : : q$  coinciden con los caracteres de texto  $T \text{CsC1} : : sCq$ , ¿cuál es el cambio mínimo  $s0 > s$  tal que para algún  $k < q$ ,

$$P 1 : : k DT \text{CEs0 C 1} : : s0 C k \quad : \quad (32.6)$$

donde  $s0 C k D s C q$ ?

En otras palabras, sabiendo que  $Pq = TsCq$ , queremos el prefijo propio más largo  $Pk$  de  $Pq$  que también sea un sufijo de  $TsCq$ . (Dado que  $s0 C k D s C q$ , si nos dan  $s$  y  $q$ , entonces encontrar el cambio más pequeño  $s0$  es equivalente a encontrar la longitud más larga del prefijo  $k$ ). Agregamos la diferencia  $q-k$  en las longitudes de estos prefijos de  $P$  a la de modo que  $s0 D s C q = k$ . En el mejor turno  $s$  para llegar a nuestro nuevo turno  $s0$ , de los casos,  $k = 0$ , de modo que  $s0 D s C q = e$  inmediatamente descartamos desplazamientos  $s C 1; s C 2; \dots; s C q - 1$ . En cualquier caso, en el nuevo turno  $s0$  no necesitamos comparar los primeros  $k$  caracteres de  $P$  con los caracteres correspondientes de  $T$  ya que la ecuación (32.6) garantiza que coincidan.

Podemos precalcular la información necesaria comparando el patrón consigo mismo, como lo demuestra la figura 32.10(c). Como  $T \text{CEs0 C 1} : : s0 C k$  es parte del

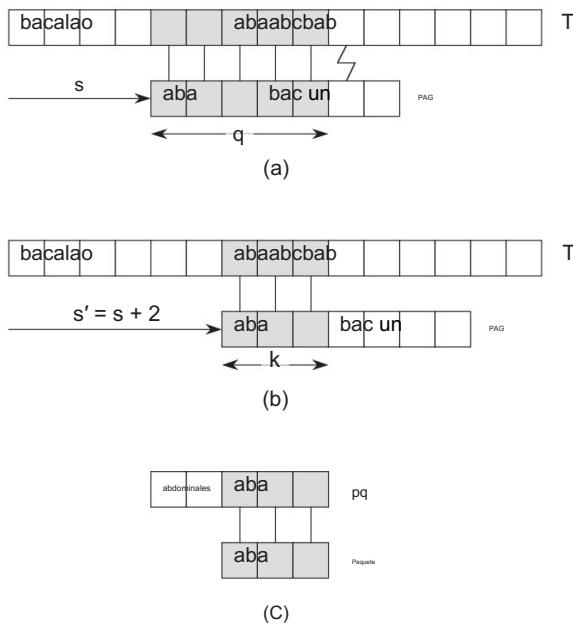


Figura 32.10 La función de prefijo. (a) El patrón PD ababaca se alinea con un texto T para que coincidan los primeros q D 5 caracteres. Los caracteres coincidentes, que se muestran sombreados, están conectados por líneas verticales. (b) Usando solo nuestro conocimiento de los 5 caracteres emparejados, podemos deducir que un cambio de s C 1 no es válido, pero que un cambio de s0 D sc2 es consistente con todo lo que sabemos sobre el texto y, por lo tanto, es potencialmente válido. (c) Podemos precalcular información útil para tales deducciones comparando el patrón consigo mismo. Aquí vemos que el prefijo más largo de P que también es un sufijo propio de P5 es P3.

Representamos esta información precalculada en el arreglo , de modo que  $\text{CE5 D } 3$ . Dado que q caracteres se han emparejado correctamente en el cambio s, el siguiente cambio potencialmente válido es en  $s0 D s Cq \text{ CEq}$  como se muestra en la parte (b).

parte conocida del texto, es un sufijo de la cadena Pq. Por lo tanto, podemos interpretar la ecuación (32.6) como pidiendo el mayor  $k < q$  tal que  $P_k = P_{q-k}$ . Entonces, el nuevo turno  $s0 D s C.qk/$  es el próximo turno potencialmente válido. Encontraremos conveniente almacenar, para cada valor de q, el número k de caracteres coincidentes en el nuevo turno  $s0$  en , lugar de almacenar, digamos,  $s0 s$ .

Formalizamos la información que precomputamos de la siguiente manera. Dado un patrón  $P \text{ CE1} : : m$ , la función de prefijo para el patrón P es la función  $W f1; 2; : : : ; mg! f0; 1; : : : ; m 1g$  tal que

$$\text{CEq D max fk W k<q y } P_{q-k} = P_k :$$

Es decir,  $\text{CEq}$  es la longitud del prefijo más largo de P que es un sufijo propio de Pq. La figura 32.11(a) da la función de prefijo completa para el patrón ababaca.

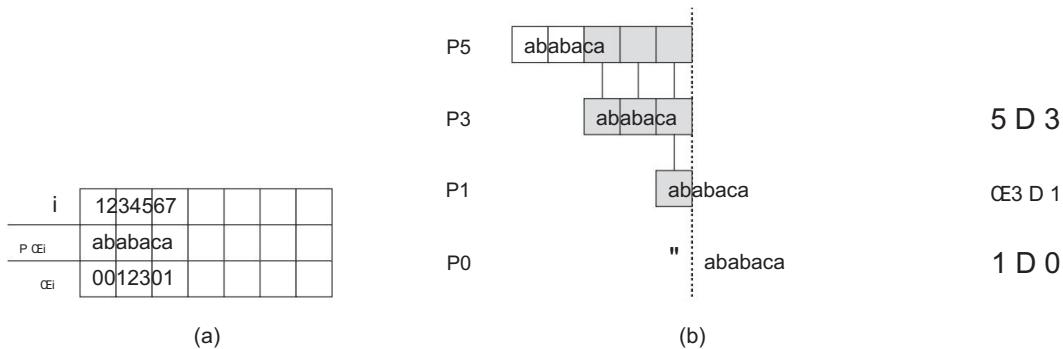


Figura 32.11 Una ilustración del Lema 32.5 para el patrón PD ababaca y q D 5. (a) La función para el patrón dado. Como 0E5 D 3, 0E3 D 1 y 0E1 D 0, iterando obtenemos 0E5 D f3; 1; 0g. (b) Deslizamos la plantilla que contiene el patrón P hacia la derecha y notamos cuando algún prefijo  $P_k$  de P coincide con algún sufijo propio de  $P_5$ ; obtenemos coincidencias cuando k D 3, 1 y 0. En la figura, la primera fila da P, y la línea vertical punteada se dibuja justo después de P5. Las filas sucesivas muestran todos los cambios de P que hacen que algún prefijo  $P_k$  de P coincida con algún sufijo de  $P_5$ . Los caracteres coincidentes con éxito se muestran sombreados. Las líneas verticales conectan los caracteres coincidentes alineados. Así, fk W k<5 y  $P_k = P_5 g$  D f3; 1; 0g. El lema 32.5 afirma que 0Eq D fk W k<q y  $P_k = P_{q+1}$  para todo q.

El pseudocódigo a continuación proporciona el algoritmo de coincidencia Knuth-Morris-Pratt como el procedimiento KMP-MATCHER. En su mayor parte, el procedimiento se sigue de FINITE-AUTOMATON-MATCHER, como veremos. KMP-MATCHER llama al procedimiento auxiliar COMPUTE-PREFIX-FUNCTION para calcular.

KMP-MATCHER.T; P / 1 n D

T:longitud 2 m D

P:longitud 3 D CÁLCULO-

PREFIJO-FUNCIÓN.P / 4 q D 0 5 para i D 1 a n

// número de caracteres coincidentes //

escanea el texto de izquierda a derecha

```

6      while q>0 y P 0Eq C 1  T 0Eq q D 0Eq si P
7          0Eq C 1 ==
8          T 0Eq q D q C 1 si q ==
9              m imprime "El
10             patrón
11             ocurre con el cambio" imq D 0Eq
12
  // el carácter siguiente no coincide
  // coincidencias del siguiente
  // carácter // ¿coincide todo P?
  // busca la siguiente coincidencia

```

COMPUTE-PREFIX-FUNCTION.P /

```

1 m D P:longitud 2 sea
Œ1 : : m un nuevo arreglo 3 Œ1 D 0 4 k
D 0 5 para q D 2
a m 6 7 8

```

```

mientras que k>0 y P Œk C 1 ≠ P Œq k D
Œk si P Œk
C 1 == P Œq k D k C 1
9      Œq D k
10
11 regreso

```

Estos dos procedimientos tienen mucho en común, porque ambos comparan una cadena con el patrón P: KMP-MATCHER compara el texto T con P, y COMPUTE PREFIX-FUNCTION compara P consigo mismo.

Comenzamos con un análisis de los tiempos de ejecución de estos procedimientos. Prueba estos procedimientos correctos serán más complicados.

#### Análisis de tiempo de ejecución

El tiempo de ejecución de COMPUTE-PREFIX-FUNCTION es  $.m/$ , que mostramos usando el método agregado de análisis amortizado (consulte la Sección 17.1). La única parte complicada es mostrar que el ciclo while de las líneas 6 y 7 ejecuta  $Om/$  veces en conjunto. Demostraremos que hace como mucho  $m 1$  iteraciones. Empezamos haciendo algunas observaciones sobre k. Primero, la línea 4 comienza con k en 0, y la única forma en que k aumenta es mediante la operación de incremento en la línea 9, que se ejecuta como máximo una vez por iteración del bucle for de las líneas 5–10. Por lo tanto, el aumento total de k es como mucho  $m1$ .

En segundo lugar, dado que  $k < q$  al ingresar al ciclo for y cada iteración del ciclo incrementa q, siempre tenemos  $k < q$ . Por lo tanto, las asignaciones en las líneas 3 y 10 aseguran que  $Œq < q$  para todo  $q D 1$ ;  $2; : : ; m$ , lo que significa que cada iteración del ciclo while disminuye k. Tercero, k nunca se vuelve negativo. Juntando estos hechos, vemos que la disminución total de k del ciclo while está acotada superiormente por el aumento total de k en todas las iteraciones del ciclo for , que es  $m 1$ .

Por lo tanto, el bucle while itera como máximo  $m 1$  veces en total, y la FUNCIÓN DE PREFIJO DE CÁLCULO se ejecuta en el tiempo  $.m/$ .

El ejercicio 32.4-4 le pide que demuestre, mediante un análisis agregado similar, que el partido El tiempo de reproducción de KMP-MATCHER es  $.n/$ .

En comparación con FINITE-AUTOMATON-MATCHER, al usar en lugar de i, hemos reducido el tiempo de preprocessamiento del patrón de  $Om |t| a .m/$ , manteniendo el tiempo de coincidencia real limitado por  $.n/$ .

### Corrección del cálculo de la función de prefijo

Veremos un poco más adelante que la función de prefijo nos ayuda a simular la función de transición  $\delta$  en un autómata de comparación de cadenas. Pero primero, necesitamos probar que el procedimiento CÁLCULO-FUNCIÓN-PREFIJO realmente calcula la función prefijo correctamente. Para hacerlo, necesitaremos encontrar todos los prefijos  $P_k$  que sean sufijos propios de un prefijo  $P_q$  dado. El valor de  $\text{Eq}$  nos da el prefijo más largo, pero el siguiente lema, ilustrado en la figura 32.11, muestra que iterando la función de prefijo podemos enumerar todos los prefijos  $P_k$  que son sufijos propios

de  $P_q$ . Sea

$$\text{Eq } D \text{ fEq; } .2/\text{Eq; } .3/\text{Eq; } \dots; .t/\text{Eq; } \text{ donde } .i /$$

$\text{Eq}$  se define en términos de iteración funcional, de modo que  $.0/\text{Eq } D q \text{ y } .i/\text{Eq } D \text{ fEq }$  para  $i < \text{Eq}$  se detiene en 1, y donde la secuencia en  $.t/\text{Eq } D 0$  alcanzando

### Lema 32.5 (Lema de iteración prefijo-función)

Sea  $P$  un patrón de longitud  $m$  con función de prefijo  $\cdot$ . Entonces, para  $q \in D$ ;  $1; \dots; m$ , tenemos  $\text{Eq } D \text{ fEq } W k < q \text{ y } P_k = P_{qg}$ .

Prueba Primero probamos que  $\text{Eq } f_k W k < q \text{ y } P_k = P_{qg}$  o, de manera equivalente,  $i < 2 \text{ Eq implica } P_i = P_q : (32.7) \text{ Eq}$ , entonces  $i \in D$ .  $u/\text{Eq}$  para alguna  $u > 0$ . Demostramos la ecuación ya que  $(32.7)$  por inducción Si  $i < 2$  sobre  $u$ . Para  $u = 1$ , tenemos  $i \in D \text{ Eq}$ , y la afirmación se deduce  $i < q$  y  $P_i = P_q$  por la definición de  $\cdot$ . Usando las relaciones  $\text{Eq } i < j \text{ y } P_i = P_j$  y la transitividad de  $<$  y establece el reclamo para todo  $i$  en  $\text{Eq}$ . Por lo tanto,  $\text{Eq } f_k W k < q \text{ y } P_k = P_{qg}$ .

Ahora demostramos que  $f_k W k < q \text{ y } P_k = P_{qg} \text{ Eq}$  por contradicción. Supongamos por el contrario que el conjunto  $f_k W k < q \text{ y } P_k = P_{qg} \text{ Eq}$  no es vacío, y sea  $j$  el número más grande del conjunto. Debido a que  $\text{Eq}$  es el valor más grande en  $f_k W k < q \text{ y } P_k = P_{qg} \text{ Eq}$ , debemos tener  $j < \text{Eq}$ , por lo que hacemos que  $j$  denote el entero más pequeño en  $\text{Eq}$  que es mayor que  $j$ . (Podemos elegir  $j \in \text{Eq}$  si ningún otro número en  $\text{Eq}$  es mayor que  $j$ ). Tenemos  $P_j = P_q$  porque  $j < 2 \text{ fEq } W k < q \text{ y } P_k = P_{qg}$ , y de  $j < \text{Eq}$  y la ecuación  $(32.7)$ , tenemos  $P_j = P_q$ . Así,  $P_j = P_{qg}$  por el Lema 32.1, y  $j$  es el mayor valor menor que  $j$  con esta propiedad. Por lo tanto, debemos tener  $\text{Eq } D j < q$ , dado que  $j$  debe tener  $j < \text{Eq}$  también. Esta contradicción confirma el lema.

$2 \text{ Eq, nosotros}$

El algoritmo CÁLCULO-PREFIJO-FUNCIÓN calcula  $\text{Eq}$ , en orden, para  $q \in D$ . Establecer  $\text{Eq}_1$  en ya que  $0$  en la línea 3 de COMPUTE-PREFIX-FUNCTION es cer  $1; 2; \dots$ ; ciertamente correcto,  $\text{Eq} < q$  para todo  $q$ . Usaremos el siguiente lema y

su corolario es probar que COMPUTE-PREFIX-FUNCTION calcula  $\text{CEq}$  correctamente para  $q > 1$ .

### Lema 32.6

Sea  $P$  un patrón de longitud  $m$ , y sea la función de prefijo para  $P$ . Para  $q \in D$ ;  $1; : : : ; m$ , si  $\text{CEq} > 0$ , entonces  $\text{CEq} 1 2 \text{CEq} 1$ .

Demostración Sea  $r \in D$   $\text{CEq} > 0$ , de modo que  $r < q$  y  $\text{Pr} = \text{Pq}$ ; por lo tanto,  $r 1 < q 1$  y  $\text{Pr} 1$

$\text{Pk} 1$  (eliminando el último carácter de  $\text{Pr}$  y  $\text{Pq}$ , lo cual podemos hacer porque  $r > 0$ ).

Por el Lema 32.5, por lo tanto,  $r 1 2 \text{CEq} 1$ . Así, tenemos  $1 \in D$   $r 1 2 \text{CEq} 1$ . ■

$\text{CEq}$

Para  $q \in D$ ;  $3; : : : ; m$ , defina el subconjunto  $\text{Eq} 1 \text{CEq} 1 \text{Dfk} 2 \text{CEq} 1 \text{WPCEkC} 1 \text{DPCEqgD}$

$\text{fk} W k < q 1 \text{y} \text{Pk} = \text{Pk} 1 \text{y} \text{PCEkC} 1 \text{DPCEqgD}$  (por el Lema 32.5)

$\text{Dfk} W k < q 1 \text{y} \text{PkC} 1 = \text{Pqg} : E$

conjunto  $\text{Eq} 1$  consta de los valores  $k < q 1$  para los cuales  $\text{Pk} = \text{Pk} 1$  y para los cuales, debido a que  $\text{PCEkC} 1 \text{DPCEqgD}$ , tenemos  $\text{PkC} 1 = \text{Pq}$ . Por lo tanto,  $\text{Eq} 1$  consta de aquellos valores  $k 2 \text{CEq} 1$  tales que podemos extender  $\text{Pk}$  a  $\text{PkC} 1$  y obtener un sufijo propio de  $\text{Pq}$ .

### Corolario 32.7

Sea  $P$  un patrón de longitud  $m$  y sea la función de prefijo para  $P$ . Para  $q \in D$ ;  $2; : : : ; m$ ,

$\text{CEq} D ; : :$

$\text{CEq} D (0^1 C \max \text{fk} 2 \text{Eq} 1 g \text{ si } \text{Eq} 1 \neq ; : )$

Prueba Si  $\text{Eq} 1$  está vacía, no hay  $k 2 \text{CEq} 1$  (incluyendo  $k \in D 0$ ) para el cual podemos extender  $\text{Pk}$  a  $\text{PkC} 1$  y obtener un sufijo adecuado de  $\text{Pq}$ . Por lo tanto  $\text{CEq} D 0$ .

Si  $\text{Eq} 1$  no está vacía, entonces para cada  $k \in \text{Eq} 1$  tenemos  $k < q$  y  $\text{PkC} 1 = \text{Pq}$ . Por lo tanto, de la definición de  $\text{CEq}$ , tenemos  $1 \in C$

$\text{CEq} \max \text{fk} 2 \text{Eq} 1 g : \text{Note} \quad (32.8)$

que  $\text{CEq} > 0$ . Sea  $r \in D$   $\text{CEq} 1$ , de modo que  $r C 1 \in D \text{CEq}$  y por lo tanto  $\text{Pr} C 1 = \text{Pq}$ . Como  $r C 1 > 0$ , tenemos  $\text{PCErC} 1 \text{DPCEq}$ . Además, por el Lema 32.6, tenemos  $r 2 \max \text{fk} 2 \text{Eq} 1 g$  o, de manera equivalente,  $\text{CEq} 1$ . Por lo tanto,  $r 2 \in \text{Eq} 1$ , entonces  $r 1 \in C \max \text{fk} 2 \text{Eq} 1 g$ : La combinación

$\text{CEq} \text{ de las ecuaciones} \quad (32.9)$

(32.8) y (32.9) completa la demostración. ■

Ahora terminamos la prueba de que COMPUTE-PREFIX-FUNCTION calcula correctamente. En el procedimiento CÁLCULO-PREFIJO-FUNCIÓN, al comienzo de cada iteración del ciclo for de las líneas 5 a 10, tenemos que  $k \leq 1$ . Esta condición se cumple en las líneas 3 y 4 cuando se ingresa al ciclo por primera vez, y sigue siendo cierto en cada iteración sucesiva debido a la línea 10. Las líneas 6–9 ajustan  $k$  para que se convierta en el valor correcto de  $\leq k$ . El ciclo while de las líneas 6 y 7 busca en todos los valores  $k \geq 1$  hasta que encuentra un valor de  $k$  para el cual  $P \leq k \leq C[1] \leq DP \leq q$ ; en ese punto,  $k$  es el valor más grande en el conjunto  $Eq_1$ , de modo que, por el corolario 32.7, podemos establecer  $\leq k \leq C[1]$ . Si el ciclo while no puede encontrar  $k \geq 1$  tal que  $P \leq k \leq C[1] \leq DP \leq q$ , entonces  $k$  es igual a 0 en la línea 8. Si  $P \leq C[1] \leq DP \leq q$ , entonces debemos establecer  $k = 1$ ; de lo contrario, deberíamos dejar  $k$  solo y establecer  $\leq k = 0$ . Las líneas 8 a 10 establecen  $k$  y  $\leq k$  correctamente en cualquier caso. Esto completa nuestra prueba de la corrección de COMPUTE PREFIX-FUNCTION.

### Corrección del algoritmo de Knuth-Morris-Pratt

Podemos pensar en el procedimiento KMP-MATCHER como una versión reimplementada del procedimiento FINITE-AUTOMATON-MATCHER, pero utilizando la función de prefijo para calcular las transiciones de estado. Específicamente, probaremos que en la  $i$ -ésima iteración de los bucles for de KMP-MATCHER y FINITE-AUTOMATON-MATCHER, el estado  $q$  tiene el mismo valor cuando probamos la igualdad con  $m$  (en la línea 10 en KMP MATCHER y en la línea 5 en FINITE-AUTOMATON-MATCHER). Una vez que hemos argumentado que KMP-MATCHER simula el comportamiento de FINITE-AUTOMATON MATCHER, la corrección de KMP-MATCHER se deriva de la corrección de FINITE-AUTOMATON-MATCHER (aunque veremos un poco más adelante por qué la línea 12 en KMP-MATCHER es necesaria).

Antes de probar formalmente que KMP-MATCHER simula correctamente FINITE AUTOMATON-MATCHER, tomemos un momento para entender cómo la función de prefijo reemplaza a la función de transición  $\delta$ . Recuerde que cuando un autómata de coincidencia de cadenas está en el estado  $q$  y escanea un carácter  $a$  en  $DT[q]$ , se mueve a un nuevo estado  $\delta(q; a)$ . Si  $a \in DP \leq C[1]$ , de modo que continúa coincidiendo con el patrón, entonces  $\delta(q; a) \in D[q] \leq C[1]$ . De lo contrario, si  $a \notin P \leq C[1]$ , de modo que  $a$  no sigue el patrón, y  $\delta(q; a) \notin D[q]$ . En el primer caso, cuando continúa coincidiendo, KMP-MATCHER pasa al estado  $q \leq C[1]$  sin hacer referencia a la función: la prueba del bucle while en la línea 6 da como resultado falso la primera vez, la prueba en la línea 8 da como resultado verdadero y línea 9 incrementos  $q$ .

La función entra en juego cuando el carácter  $a$  no sigue coincidiendo con el patrón, por lo que el nuevo estado  $\delta(q; a)$  es  $q$  o está a la izquierda de  $q$  a lo largo de la columna vertebral del autómata. El ciclo while de las líneas 6–7 en KMP-MATCHER itera a través de los estados en  $Eq_1$ , deteniéndose cuando llega a un estado, digamos  $q_0$ , tal que  $a$  coincide con  $P \leq q_0 \leq C[1]$  o  $q_0$  ha bajado hasta 0. Si  $a$  coincide con  $P \leq q_0 \leq C[1]$ ,

luego, la línea 9 establece el nuevo estado en  $q_0 C_1$ , que debería ser igual a  $i.q; a/$  para que la simulación funcione correctamente. En otras palabras, el nuevo estado  $i.q; a/$  debe ser el estado 0 o uno mayor que algún estado en  $Ceq$ .

Veamos el ejemplo de las Figuras 32.7 y 32.11, que son para el patrón PD ababaca. Supongamos que el autómata está en el estado  $q D 5$ ; los estados en  $C5$  son, en orden descendente, 3, 1 y 0. Si

el siguiente carácter escaneado es c, podemos ver fácilmente que el autómata se mueve al estado  $i.5; c/ D 6$  tanto en FINITE AUTOMATON-MATCHER como en KMP-MATCHER.

Supongamos ahora que el siguiente carácter explorado es b, de modo que el autómata debería pasar al estado  $i.5; b/D 4$ .

El ciclo while en KMP-MATCHER sale habiendo ejecutado la línea 7 una vez, y llega al estado  $q_0 D C5 D 3$ . Como  $P Ceq_0 C 1 DP C4 D b$ , la prueba en la línea 8 se cumple y KMP-MATCHER se mueve al nuevo estado  $q_0 C 1 D 4 D i.5; b/$ .

Finalmente, supongamos que el siguiente carácter escaneado es en cambio a, por lo que el autómata debe pasar al estado  $i.5; a/ D 1$ . Las primeras tres veces que se ejecuta la prueba en la línea 6, la prueba resulta verdadera. La primera vez, encontramos que  $P C6 D c \neq a$ , y KMP-MATCHER se mueve al estado  $C5 D 3$  (el primer estado en  $C5$ ). La segunda vez, encontramos que  $P C4 D b \neq a$  y pasamos al estado  $C3 D 1$  (el segundo estado en  $C5$ ). La tercera vez, encontramos que  $P C2 D b \neq a$  y pasamos al estado  $C1 D 0$  (el último estado en  $C5$ ). El ciclo while sale una vez que llega al estado  $q_0 D 0$ . Ahora, la línea 8 encuentra que  $P Ceq_0 C 1 DP C1 D a$ , y la línea 9 mueve el autómata al nuevo estado  $q_0 C 1 D 1 D i.5; a/$ .

Por lo tanto, nuestra intuición es que KMP-MATCHER itera a través de los estados en  $Ceq$  en orden decreciente, deteniéndose en algún estado  $q_0$  y luego posiblemente moviéndose al estado  $q_0 C_1$ . Aunque eso puede parecer mucho trabajo solo para simular la computación  $i.q; a/$ , tenga en cuenta que asintóticamente, KMP-MATCHER no es más lento que FINITE AUTOMATON-MATCHER.

Ahora estamos listos para probar formalmente la exactitud del algoritmo de Knuth-Morris-Pratt. Por el Teorema 32.4, tenemos que  $q D .Ti/$  después de cada vez que ejecutamos la línea 4 de FINITE-AUTOMATON-MATCHER. Por lo tanto, basta con mostrar que se cumple la misma propiedad con respecto al bucle for en KMP-MATCHER. La prueba procede por inducción sobre el número de iteraciones del ciclo. Inicialmente, ambos procedimientos establecen  $q$  en 0 cuando ingresan sus respectivos bucles for por primera vez. Considere la iteración  $i$  del ciclo for en KMP-MATCHER, y deje que  $q_0$  sea el estado al comienzo de esta iteración del ciclo. Por la hipótesis inductiva, tenemos  $q_0 D .Ti1/$ . Necesitamos demostrar que  $q D .Ti/$  en la línea 10. (Nuevamente, manearemos la línea 12 por separado).

Cuando consideramos el carácter  $T Cei$ , el prefijo más largo de  $P$  que es un sufijo de  $Ti$  es  $Pq_0C_1$  (si  $P Ceq_0 C 1 DT Cei$ ) o algún prefijo (no necesariamente propio y posiblemente vacío) de  $Pq_0$ . Consideramos por separado los tres casos en los que  $.Ti/ D 0$ ,  $.Ti/ D q_0 C 1 y 0 < .Ti/ q_0$ .

Si  $.Ti / D \neq 0$ , entonces  $P_0 D$  " es el único prefijo de  $P$  que es un sufijo de  $T_i$  . El ciclo while de las líneas 6–7 itera a través de los valores en  $\{Eq_0\}$ , pero aunque  $Pq = Ti$  para cada  $q \in \{Eq_0\}$ , el bucle nunca encuentra  $q$  tal que  $P \neq C_1 DT$   $\neq i$ . El bucle termina cuando  $q$  llega a 0 y, por supuesto, no se ejecuta la línea 9. Por lo tanto,  $q = 0$  en la línea 10, de modo que  $q = D \cdot Ti$ .

Si  $.Ti / D \neq q_0 C_1$ , entonces  $P \neq q_0 C_1 DT$   $\neq i$ , y la prueba del ciclo while en la línea 6 falla la primera vez. La línea 9 se ejecuta, incrementando  $q$  de modo que luego tengamos  $q = D \cdot q_0 C_1 D \cdot Ti$ .

Si  $0 < .Ti / q_0$ , entonces el ciclo while de las líneas 6–7 itera al menos una vez, verificando en orden decreciente cada valor  $q \in \{Eq_0\}$  hasta que se detiene en algún  $q < q_0$ .

Por lo tanto,  $Pq$  es el prefijo más largo de  $Pq_0$  para el cual  $P \neq C_1 DT$   $\neq i$ , de modo que cuando termina el ciclo while,  $q = C_1 D \cdot Pq_0 T$   $\neq i$ . Como  $q = D \cdot Ti$ , el Lema 32.3 implica que  $.Ti_1 T \neq i / D \cdot Pq_0 T$   $\neq i$ . Así, tenemos

$$\begin{aligned} q &= C_1 D \cdot Pq_0 T \neq i / D \cdot Ti_1 \\ &\neq i / D \cdot Ti \end{aligned}$$

cuando termina el bucle while . Despues de que la línea 9 incrementa  $q$ , tenemos  $q = D \cdot Ti$ .

La línea 12 es necesaria en KMP-MATCHER porque, de lo contrario, podríamos hacer referencia a  $P \neq C_1$  en la línea 6 después de encontrar una ocurrencia de  $P$ . (El argumento de que  $q = D \cdot Ti$  / en la siguiente ejecución de la línea 6 sigue siendo válido por el sugerencia dada en el ejercicio 32.4-8:  $i.m; a / D i.C_1; a / o$ , de manera equivalente,  $.P a / D \cdot P \neq C_1$  para cualquier  $a \in \Sigma$ .) El argumento restante para la corrección de Knuth -El algoritmo de Morris Pratt se deriva de la corrección de FINITE-AUTOMATON-MATCHER, ya que hemos demostrado que KMP-MATCHER simula el comportamiento de FINITE AUTOMATON-MATCHER.

### Ejercicios

#### 32.4-1

Calcule la función de prefijo para el patrón ababbabbabbabbabbabb.

#### 32.4-2

Proporcione un límite superior para el tamaño de  $Eq$  en función de  $q$ . Da un ejemplo para mostrar que tu límite es estrecho.

#### 32.4-3

Explique cómo determinar las ocurrencias del patrón  $P$  en el texto  $T$  examinando la función para la cadena  $PT$  (la cadena de longitud  $m+n$  que es la concatenación de  $P$  y  $T$ ).

## 32.4-4

Use un análisis agregado para mostrar que el tiempo de ejecución de KMP-MATCHER es  $\sim n$ .

## 32.4-5

Use una función potencial para mostrar que el tiempo de ejecución de KMP-MATCHER es  $\sim n$ .

## 32.4-6

Muestre cómo mejorar KMP-MATCHER reemplazando la aparición de en la línea 7 (pero no en la línea 12) por se define  $\delta_D$  recursivamente para  $q = D_1; D_2; \dots; D_m$  por la ecuación

$$\begin{aligned} \delta_D &= 0 && \text{si } D = 0 \\ &= \delta_{D'} && \text{si } D' \neq 0 \text{ y } P[D] = C[1] \text{ DP}[C] = 1 \text{ y } \delta_{D'} = 0 \\ &= 0 && \text{si } P[D] \neq C[1] \end{aligned}$$

Explique por qué el algoritmo modificado es correcto, y explique en qué sentido este cambio constituye una mejora.

## 32.4-7

Dé un algoritmo de tiempo lineal para determinar si un texto  $T$  es una rotación cíclica de otra cadena  $T'$ . Por ejemplo, arc y car son rotaciones cíclicas entre sí.

## 32.4-8 ?

Proporcione un  $O(m|T|)$ -algoritmo de tiempo para calcular la función de transición  $\delta$  para el autómata de emparejamiento de cadenas correspondiente a un patrón dado  $P$ . (Sugerencia: Demuestre que  $\delta(q, a) = D_i$  si  $a = D_i$  y  $\delta(q, a) = \delta(D_i, a)$  para alguna  $i$ .)

## Problemas

## 32-1 Coincidencia de cadenas basada en factores de

repetición Sea  $y$  la concatenación de la cadena  $x$  consigo misma  $r$  veces. Por ejemplo, .ab/3D ababab. Decimos que una cadena  $x$  tiene factor de repetición  $r$  si  $x = y^r$  para alguna cadena  $y$  y alguna  $r > 0$ . Sea  $x/r$  el mayor  $r$  tal que  $x$  tiene un factor de repetición  $r$ .

- a. Proporcione un algoritmo eficiente que tome como entrada un patrón  $P$  de  $m$  caracteres y calcule el valor  $\delta_P(i)$  para  $i = 1; 2; \dots; m$ . ¿Cuál es el tiempo de ejecución de su algoritmo?

- b. Para cualquier patrón  $P \in \{0,1\}^m$ , defina  $\text{Pf} /$  como  $\max_{i=1}^m \text{Pi}_i$ . Demuestre que si el patrón  $P$  se elige aleatoriamente del conjunto de todas las cadenas binarias de longitud  $m$ , entonces el valor esperado de  $\text{Pf}$  es  $0.1^m$ .
- C. Argumente que el siguiente algoritmo de emparejamiento de cadenas encuentra correctamente todas las ocurrencias del patrón  $P$  en un texto  $T \in \{0,1\}^n$  en el tiempo  $O(\text{Pf} / n C m)$ :

#### REPETICIÓN-COMPARADOR.P; T / 1

```

m D P:longitud 2 n D
T:longitud 3 k D 1
C .P / 4 q D 0 5 s D 0 6
while snm 7
  si T <= C q
    C 1 == P <= C 1 8 q D q
    C 1 9 si q == m 10 imprime "El patrón ocurre
    con el cambio" s si q == m o
    T <= C q C 1 ≠ P <= C 1
    s D s C max.1; dq=ke/ q D 0
11
12
13

```

Este algoritmo se debe a Galil y Seiferas. Al ampliar estas ideas en gran medida, obtuvieron un algoritmo de coincidencia de cadenas de tiempo lineal que usa solo  $0.1^m /$  de almacenamiento más allá de lo que se requiere para  $P$  y  $T$ .

#### Notas del capítulo

Aho, Hopcroft y Ullman [5] analizan la relación entre la coincidencia de cadenas y la teoría de los autómatas finitos. El algoritmo Knuth-Morris-Pratt [214] fue inventado de forma independiente por Knuth y Pratt y por Morris; publicaron su trabajo conjuntamente. Reingold, Urban y Gries [294] ofrecen un tratamiento alternativo del algoritmo de Knuth-Morris-Pratt. El algoritmo de Rabin-Karp fue propuesto por Karp y Rabin [201]. Galil y Seiferas [126] proporcionan un interesante algoritmo determinista de coincidencia de cadenas de tiempo lineal que usa solo  $0.1^m /$  espacio más allá del requerido para almacenar el patrón y el texto.

---

## 33

# Geometría Computacional

La geometría computacional es la rama de la informática que estudia algoritmos para resolver problemas geométricos. En la ingeniería y las matemáticas modernas, la geometría computacional tiene aplicaciones en campos tan diversos como los gráficos por computadora, la robótica, el diseño VLSI, el diseño asistido por computadora, el modelado molecular, la metalurgia, la fabricación, el diseño textil, la silvicultura y la estadística. La entrada a un problema de geometría computacional suele ser una descripción de un conjunto de objetos geométricos, como un conjunto de puntos, un conjunto de segmentos de línea o los vértices de un polígono en el sentido contrario a las agujas del reloj. El resultado suele ser una respuesta a una consulta sobre los objetos, como si alguna de las líneas se interseca, o quizás un nuevo objeto geométrico, como el casco convexo (el polígono convexo más pequeño que lo encierra) del conjunto de puntos.

En este capítulo, analizamos algunos algoritmos de geometría computacional en dos dimensiones, es decir, en el plano. Representamos cada objeto de entrada por un conjunto de puntos  $p_1; p_2; p_3; \dots; p_g$ , donde cada  $p_i$  es  $D .x_i; y_i/$  y  $x_i; y_i \in R$ . Por ejemplo, representamos un polígono  $P$  de  $n$  vértices mediante una secuencia  $hp_0; p_1; p_2; \dots; p_{n-1}$  de sus vértices en orden de aparición en el límite de  $P$ . La geometría computacional también se puede aplicar a tres dimensiones, e incluso a espacios de dimensiones superiores, pero estos problemas y sus soluciones pueden ser muy difíciles de visualizar. Incluso en dos dimensiones, sin embargo, podemos ver una buena muestra de técnicas de geometría computacional.

La sección 33.1 muestra cómo responder preguntas básicas sobre segmentos de línea de manera eficiente y precisa: si un segmento está en el sentido de las agujas del reloj o en el sentido contrario a las agujas del reloj desde otro que comparte un punto final, en qué dirección giramos al atravesar dos segmentos de línea contiguos y si dos segmentos de línea se intersecan. La sección 33.2 presenta una técnica llamada "barido" que usamos para desarrollar un algoritmo  $O(n \lg n)$ -time para determinar si un conjunto de  $n$  segmentos de línea contiene alguna intersección. La sección 33.3 proporciona dos algoritmos de "barido rotacional" que calculan el casco convexo (el polígono convexo más pequeño que lo encierra) de un conjunto de  $n$  puntos: el escaneo de Graham, que se ejecuta en el tiempo en  $\lg n$ , y la marcha de Jarvis, que toma  $O(nh)$  tiempo, donde  $h$  es el número de vértices del casco convexo. Finalmente, la Sección 33.4 da

un algoritmo On  $\lg n$ -time divide y vencerás para encontrar el par de puntos más cercano en un conjunto de  $n$  puntos en el plano.

### 33.1 Propiedades del segmento de línea

Varios de los algoritmos de geometría computacional de este capítulo requieren respuestas a preguntas sobre las propiedades de los segmentos de línea. Una combinación convexa de dos puntos distintos  $p_1 \in D_{x_1, y_1}$  y  $p_2 \in D_{x_2, y_2}$  es cualquier punto  $p_3 \in D_{x_3, y_3}$  tal que para algún  $\lambda$  en el rango  $0 < \lambda < 1$ , tenemos  $x_3 = \lambda x_1 + (1 - \lambda)x_2$  y  $y_3 = \lambda y_1 + (1 - \lambda)y_2$ . También escribimos que  $p_3 \in p_1 p_2$ . Intuitivamente,  $p_3$  es cualquier punto que está en la línea que pasa por  $p_1$  y  $p_2$  y está en o entre  $p_1$  y  $p_2$  en la línea. Dados dos puntos distintos  $p_1$  y  $p_2$ , el segmento de línea  $p_1 p_2$  es el conjunto de combinaciones convexas de  $p_1$  y  $p_2$ . Llamamos a  $p_1$  y  $p_2$  los extremos del segmento  $p_1 p_2$ . A veces importa el orden de  $p_1$  y  $p_2$ , y hablamos del segmento dirigido  $p_1 p_2$ . Si  $p_1$  es el origen  $(0, 0)$ , entonces podemos tratar el segmento  $p_1 p_2$  como el vector  $p_2$ .

En esta sección, exploraremos las siguientes preguntas:

1. Dados dos segmentos dirigidos  $p_0 p_1$  y  $p_0 p_2$ , es  $p_0 p_1$  ! en el sentido de las agujas del reloj desde  $p_0 p_2$  con respecto a su punto final común  $p_0$ ?
2. Dados dos segmentos de línea  $\overline{p_0 p_1}$  y  $\overline{p_1 p_2}$ , si recorremos  $\overline{p_0 p_1}$  y luego  $\overline{p_1 p_2}$ , ¿doblamos a la izquierda en el punto  $p_1$ ?
3. ¿Se cruzan los segmentos de línea  $p_1 p_2$  y  $p_3 p_4$  ?

No hay restricciones en los puntos dados.

Podemos responder cada pregunta en tiempo  $O(1)$ , lo que no debería sorprender ya que el tamaño de entrada de cada pregunta es  $O(1)$ . Además, nuestros métodos utilizan únicamente sumas, restas, multiplicaciones y comparaciones. No necesitamos división ni funciones trigonométricas, las cuales pueden ser computacionalmente costosas y propensas a problemas con errores de redondeo. Por ejemplo, el método “sencillo” para determinar si dos segmentos se intersecan: calcular la ecuación lineal de la forma  $y = mx + b$  para cada segmento ( $m$  es la pendiente y  $b$  es la intersección con el eje  $y$ ), encontrar el punto de intersección de las líneas y verifique si este punto está en ambos segmentos; usa la división para encontrar el punto de intersección. Cuando los segmentos son casi paralelos, este método es muy sensible a la precisión de la operación de división en computadoras reales. El método de esta sección, que evita la división, es mucho más preciso.

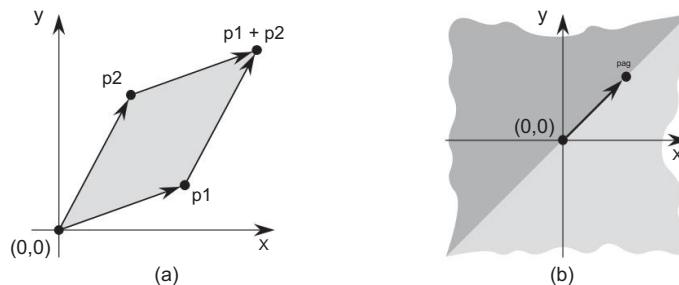


Figura 33.1 (a) El producto vectorial de los vectores  $p_1$  y  $p_2$  es el área con signo del paralelogramo. (b) La región ligeramente sombreada contiene vectores que van en el sentido de las agujas del reloj desde  $p$ . La región sombreada oscura contiene vectores que van en sentido antihorario desde  $p$ .

### productos cruzados

El cálculo de productos cruzados se encuentra en el corazón de nuestros métodos de segmento de línea. Considere los vectores  $p_1$  y  $p_2$ , que se muestran en la figura 33.1(a). Podemos interpretar el producto vectorial  $p_1 \cdot p_2$  como el área con signo del paralelogramo formado por los puntos  $.0; 0/$ ,  $p_1$ ,  $p_2$  y  $p_1 \cdot p_2 D .x_1 C x_2; y_1 C y_2/$ . Una definición equivalente, pero más útil, da el producto vectorial como el determinante de una matriz:<sup>1</sup>

$$\begin{array}{c} p_1 \cdot p_2 = \det \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \\ \end{vmatrix} \\ = \frac{x_1 y_2 - x_2 y_1}{2} \end{array}$$

Si  $p_1 \cdot p_2$  es positivo, entonces  $p_1$  es en el sentido de las manecillas del reloj desde  $p_2$  con respecto al origen  $.0; 0/$ ; si este producto vectorial es negativo, entonces  $p_1$  es en sentido antihorario desde  $p_2$ . (Véase el ejercicio 33.1-1.) La figura 33.1(b) muestra las regiones en el sentido de las agujas del reloj y en el sentido contrario a las agujas del reloj en relación con un vector  $p$ . Surge una condición de contorno si el producto cruzado es 0; en este caso, los vectores son colineales y apuntan en la misma dirección o en direcciones opuestas.

Para determinar si un segmento dirigido  $p_0 p$  está más cerca de un segmento dirigido  $p_0 p$  a su punto final  $p_2$  en el sentido de las agujas del reloj o en el sentido contrario a las agujas del reloj con respecto común  $p_0$ , simplemente traducimos para usar  $p_0$  como el origen. Es decir, hacemos que  $p_1 \cdot p_0$  denote el vector  $p_0 D .x_0 D x_1 x_0 y_0 D y_1 y_0$ , y definimos  $p_2 \cdot p_0$  de manera similar. Luego calculamos el producto cruz

<sup>1</sup>En realidad, el producto vectorial es un concepto tridimensional. Es un vector que es perpendicular tanto a  $p_1$  como a  $p_2$  según la “regla de la mano derecha” y cuya magnitud es  $|x_1 y_2 - x_2 y_1|$ . En este capítulo, sin embargo, encontramos conveniente tratar el producto cruz simplemente como el valor  $x_1 y_2 - x_2 y_1$ .

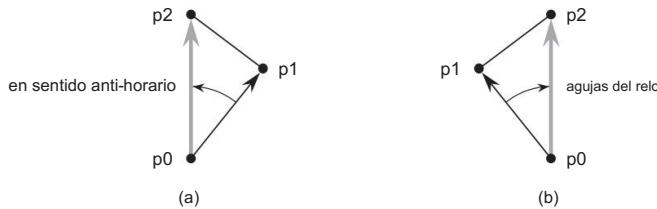


Figura 33.2 Usar el producto vectorial para determinar cómo los segmentos de línea consecutivos  $\overrightarrow{p_0p_1}$  y  $\overrightarrow{p_1p_2}$  giran en el punto  $p_1$ . Comprobamos si el segmento dirigido  $p_0 \overrightarrow{p_1}$  es en sentido horario o antihorario en relación con el segmento dirigido  $p_0 \overrightarrow{p_2}$ .  
 (a) Si es en sentido antihorario, los puntos giran a la izquierda. (b) Si giran en el sentido de las agujas del reloj, giran a la derecha.

$.p_1\ p_0/\ .p_2\ p_0/\ D\ .x_1\ x_0/.y_2\ y_0/\ .x_2\ x_0/.y_1\ y_0/ :$

Si este producto vectorial es positivo, entonces  $p_0 \overrightarrow{p_1}$  es en el sentido de las manecillas del reloj desde  $p_0 \overrightarrow{p_2}$ ; si es negativo, es en sentido antihorario.

#### Determinar si los segmentos consecutivos giran a la izquierda o a la derecha

Nuestra siguiente pregunta es si dos segmentos de línea consecutivos  $\overrightarrow{p_0p_1}$  y  $\overrightarrow{p_1p_2}$  giran a la izquierda o a la derecha en el punto  $p_1$ . De manera equivalente, queremos un método para determinar en qué dirección gira un ángulo dado  $\angle p_0p_1p_2$ . Los productos cruzados nos permiten responder esta pregunta sin calcular el ángulo. Como muestra la figura 33.2, simplemente comprobamos si el segmento dirigido  $p_0 \overrightarrow{p_1}$  está en el sentido de las agujas del reloj o en el sentido contrario a las agujas del reloj en relación con el segmento dirigido  $p_0 \overrightarrow{p_2}$ . Para hacerlo, calculamos el producto vectorial  $.p_2\ p_0/\ .p_1\ p_0/$ . Si el signo de este producto vectorial es negativo, entonces  $p_0 \overrightarrow{p_2}$  es contrario a las manecillas del reloj con respecto a  $p_0 \overrightarrow{p_1}$  y por lo tanto giramos a la izquierda en  $p_1$ . Un producto cruzado positivo indica una orientación en el sentido de las agujas del reloj y un giro a la derecha. Un producto vectorial de 0 significa que los puntos  $p_0$ ,  $p_1$  y  $p_2$  son colineales.

#### Determinar si dos segmentos de línea se intersecan

Para determinar si dos segmentos de línea se intersecan, verificamos si cada segmento se extiende a ambos lados de la línea que contiene al otro. Un segmento  $\overrightarrow{p_1p_2}$  se extiende a ambos lados de una línea si el punto  $p_1$  se encuentra en un lado de la línea y el punto  $p_2$  se encuentra en el otro lado. Surge un caso de frontera si  $p_1$  o  $p_2$  se encuentran directamente sobre la línea. Dos segmentos de línea se intersecan si y solo si se cumple una (o ambas) de las siguientes condiciones:

1. Cada segmento se extiende a ambos lados de la línea que contiene al otro.
2. Un extremo de un segmento se encuentra en el otro segmento. (Esta condición proviene del caso límite).

Los siguientes procedimientos implementan esta idea. SEGMENTOS-INTERSECCIÓN devuelve VERDADERO si los segmentos  $p_1p_2$  y  $p_3p_4$  se cruzan y FALSO si no lo hacen. Llama a las subrutinas DIRECCIÓN, que calcula las orientaciones relativas usando el método de producto cruzado anterior, y EN SEGMENTO, que determina si un punto que se sabe que es colineal con un segmento se encuentra en ese segmento.

SEGMENTOS-INTERSECCIÓN.p1; p2; p3;

```

p4/ 1 d1 D DIRECCIÓN.p3; p4; p1/
2 d2 D DIRECCIÓN.p3; p4; p2/ 3 d3
D DIRECCIÓN.p1; p2; p3/ 4 d4 D
DIRECCIÓN.p1; p2; p4/ 5 si ..d1 > 0
y d2 < 0/ o .d1 < 0 y d2 > 0// y ..d3 > 0 y d4 < 0/ o .d3 < 0 y
d4 > 0// devuelve TRUE
```

```

6 7 elseif d1 == 0 y ON-SEGMENT.p3; p4; p1/ 8
devuelve TRUE 9 elseif
```

```

d2 == 0 and ON-SEGMENT.p3; p4; p2/ 10 devuelve
VERDADERO
```

```

11 elseif d3 == 0 y ON-SEGMENT.p1; p2; p3/
12           devolver VERDADERO
```

```

13 elseif d4 == 0 y ON-SEGMENT.p1; p2; p4/
14           devolver VERDADERO
```

```

15 de lo contrario devuelve FALSO
```

DIRECCIÓN.pi; pj ; pk/

1 devuelve .pk pi/ .pj pi/

EN SEGMENTO.pi ;

pj ; pk/ 1 si min.xi ; xj / xk máx.xi; xj / y min.yi; yj / yk max.yi; yj /

2 devuelve VERDADERO 3 de lo

contrario devuelve FALSO

SEGMENTS-INTERSECT funciona de la siguiente manera. Las líneas 1 a 4 calculan la orientación relativa  $d_i$  de cada extremo  $p_i$  con respecto al otro segmento. Si todas las orientaciones relativas son distintas de cero, podemos determinar fácilmente si los segmentos  $p_1p_2$  y  $p_3p_4$  se intersecan, de la siguiente manera. El segmento  $p_1p_2$  se extiende a ambos lados de la línea que contiene el segmento  $p_3p_4$  si se dirige a los segmentos  $p_1p_2$  y  $p_3p_4$ . Los segmentos  $p_1p_2$  y  $p_3p_4$  tienen orientaciones opuestas relativas lados  $p_3p_4$ . En este caso, los signos de  $d_1$  y  $d_2$  difieren. De manera similar,  $p_3p_4$  se extiende a ambos de  $p$  la línea que contiene  $p_1p_2$  si los signos de  $d_3$  y  $d_4$  difieren. Si la prueba de la línea 5 es verdadera, entonces los segmentos se extienden entre sí y SEGMENTOS-INTERSECCIÓN devuelve VERDADERO. La figura 33.3(a) muestra este caso. De lo contrario, los segmentos no se extienden a horcajadas

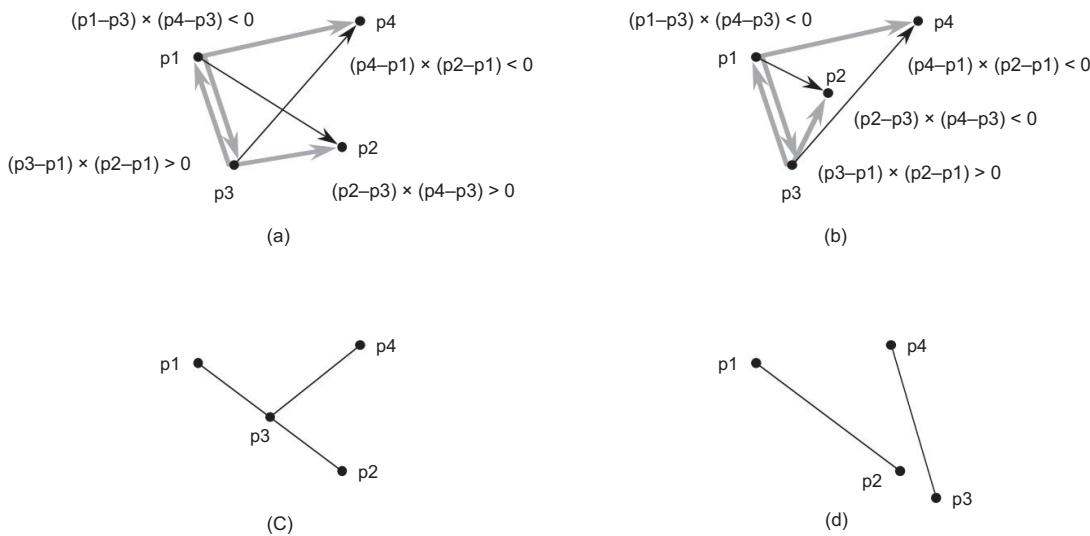


Figura 33.3 Casos en el procedimiento SEGMENTOS-INTERSECCIÓN. (a) Los segmentos  $\overline{p_1p_2}$  y  $\overline{p_3p_4}$  están a ambos lados de las líneas del otro. Debido a que  $p_3p_4$  se extiende a ambos lados de la línea que contiene a  $\overline{p_1p_2}$ , los signos de los productos cruzados  $.p_3\ p_1/ .p_2\ p_1/ .p_4\ p_1/ .p_2\ p_1/$  difieren. Debido a que  $p_1p_2$  se extiende a ambos lados de la línea que contiene  $p_3p_4$ , los signos de los productos cruzados  $.p_1\ p_3/ .p_4\ p_3/ .p_2\ p_3/ .p_4\ p_3/$  difieren. (b) El segmento  $\overline{p_3p_4}$  se extiende a ambos lados de la línea que contiene a  $\overline{p_1p_2}$ , pero  $p_1p_2$  no se extiende a ambos lados de la línea que contiene a  $p_3p_4$ . Los signos de los productos cruzados  $.p_1\ p_3/ .p_4\ p_3/ .p_2\ p_3/ .p_4\ p_3/$  son iguales. (c) El punto  $p_3$  es colineal con  $p_1p_2$  y está entre  $p_1$  y  $p_2$ . (d) El punto  $p_3$  es colineal con  $p_1p_2$ , pero no está entre  $p_1$  y  $p_2$ . Los segmentos no se intersecan.

las líneas de cada uno, aunque se puede aplicar un caso límite. Si todas las orientaciones relativas son distintas de cero, no se aplica ningún caso límite. Todas las pruebas contra 0 en las líneas 7 a 13 fallan y SEGMENTOS-INTERSECCIÓN devuelve FALSO en la línea 15. La figura 33.3(b) muestra este caso.

Se produce un caso límite si cualquier orientación relativa  $dk$  es 0. Aquí, sabemos que  $p_k$  es colineal con el otro segmento. Está directamente sobre el otro segmento si y sólo si está entre los extremos del otro segmento. El procedimiento ON-SEGMENT devuelve si  $p_k$  está entre los extremos del segmento  $p_i p_j$ , que será el otro segmento cuando se llame en las líneas 7–13; el procedimiento asume que  $p_k$  es colineal con el segmento  $p_i p_j$ . Las figuras 33.3(c) y (d) muestran casos con puntos colineales. En la figura 33.3(c),  $p_3$  está en  $p_1p_2$ , por lo que SEGMENTOS-INTERSECCIÓN devuelve VERDADERO en la línea 12. No hay puntos finales en otros segmentos en la figura 33.3(d), por lo que SEGMENTOS INTERSECCIÓN devuelve FALSO en la línea 15.

### Otras aplicaciones de los productos cruzados

Las secciones posteriores de este capítulo introducen usos adicionales para los productos cruzados.

En la Sección 33.3, necesitaremos clasificar un conjunto de puntos según sus ángulos polares con respecto a un origen dado. Como se le pide que muestre en el ejercicio 33.1-3, podemos usar productos cruzados para realizar las comparaciones en el procedimiento de clasificación. En la Sección 33.2, utilizaremos árboles rojo-negro para mantener el ordenamiento vertical de un conjunto de segmentos de línea. En lugar de mantener valores clave explícitos que comparamos entre sí en el código de árbol rojo-negro, calcularemos un producto cruzado para determinar cuál de los dos segmentos que intersecan una línea vertical dada está por encima del otro.

### Ejercicios

#### 33.1-1

Demuestre que si  $p_1 \cdot p_2$  es positivo, entonces el vector  $p_1$  está en el sentido de las manecillas del reloj desde el vector  $p_2$  con respecto al origen  $.0; 0/$  y que si este producto vectorial es negativo, entonces  $p_1$  es en sentido antihorario desde  $p_2$ .

#### 33.1-2

El profesor van Pelt propone que solo se necesita probar la dimensión x en la línea 1 de ON SEGMENT. Mostrar por qué el profesor está equivocado.

#### 33.1-3

El ángulo polar de un punto  $p_1$  con respecto a un punto de origen  $p_0$  es el ángulo del vector  $p_1 \cdot p_0$  en el sistema de coordenadas polares habitual. Por ejemplo, el ángulo polar de  $.3; 5/$  con respecto a  $.2; 4/$  es el ángulo del vector  $.1; 1/$ , que es 45 grados o  $= 4$  radianes. El ángulo polar de  $.3; 3/$  con respecto a  $.2; 4/$  es el ángulo del vector  $.1; 1/$ , que son 315 grados o  $= 7=4$  radianes. Escribir pseudocódigo para ordenar una secuencia  $hp_1; p_2; \dots; p_n$  de  $n$  puntos según sus ángulos polares con respecto a un punto origen dado  $p_0$ . Su procedimiento debe tomar  $O(n \lg n)$  tiempo y usar productos cruzados para comparar ángulos.

#### 33.1-4

Muestre cómo determinar en  $O(n^2 \lg n)$  tiempo si tres puntos cualesquiera en un conjunto de  $n$  puntos son colineales.

#### 33.1-5

Un polígono es una curva cerrada lineal por tramos en el plano. Es decir, es una curva que termina en sí misma y que está formada por una secuencia de segmentos de línea recta, llamados lados del polígono. Un punto que une dos lados consecutivos es un vértice del polígono. Si el polígono es simple, como supondremos generalmente, no se cruza a sí mismo.

El conjunto de puntos en el plano encerrado por un polígono simple forma el interior de

el polígono, el conjunto de puntos en el mismo polígono forma su límite, y el conjunto de puntos que rodean al polígono forma su exterior. Un polígono simple es convexo si, dados dos puntos cualesquiera en su límite o en su interior, todos los puntos del segmento de línea trazado entre ellos están contenidos en el límite o interior del polígono.

Un vértice de un polígono convexo no puede expresarse como una combinación convexa de dos puntos distintos en el límite o en el interior del polígono.

El profesor Amundsen propone el siguiente método para determinar si una sucesión  $hp0; p1; \dots; pn1$  de  $n$  puntos forma los vértices consecutivos de un polígono convexo. Salida "sí" si el conjunto  $\{p_i\}_{i=0}^n$  contiene giros a la izquierda ni giros a la derecha; de lo contrario, emite "no". Demuestre que aunque este método se ejecuta en tiempo lineal, no siempre produce la respuesta correcta. Modifique el método del profesor para que siempre produzca la respuesta correcta en tiempo lineal.

### 33.1-6

Dado un punto  $p0 D .x0; y0/$ , el rayo horizontal derecho desde  $p0$  es el conjunto de puntos  $fpi D .xi; yi/$   $xi > x0$  y  $yi > y0$ , es decir, es el conjunto de puntos por la derecha de  $p0$  junto con el propio  $p0$ . Muestre cómo determinar si un rayo horizontal derecho dado de  $p0$  interseca un segmento de línea  $p1p2$  en  $O(1)$  tiempo reduciendo el problema a determinar si dos segmentos de línea se intersecan.

### 33.1-7

Una forma de determinar si un punto  $p0$  está en el interior de un polígono  $P$  simple, pero no necesariamente convexo, es mirar cualquier rayo desde  $p0$  y verificar que el rayo interseca el límite de  $P$  un número impar de veces pero que  $p0$  en sí mismo no está en el límite de  $P$ . Muestre cómo calcular en tiempo  $O(n)$  si un punto  $p0$  está en el interior de un polígono  $P$  de  $n$  vértices. (Sugerencia: utilice el ejercicio 33.1-6. Asegúrese de que su algoritmo es correcto cuando el rayo interseca el límite del polígono en un vértice y cuando el rayo se superpone a un lado del polígono).

### 33.1-8

Muestre cómo calcular el área de un polígono simple, pero no necesariamente convexo, de  $n$  vértices en un tiempo  $O(n)$ . (Consulte el Ejercicio 33.1-5 para conocer las definiciones relacionadas con los polígonos).

## 33.2 Determinar si algún par de segmentos se interseca

Esta sección presenta un algoritmo para determinar si dos segmentos de línea en un conjunto de segmentos se cruzan. El algoritmo utiliza una técnica conocida como "barriendo", que es común a muchos algoritmos de geometría computacional. Además, como

Los ejercicios al final de esta sección muestran que este algoritmo, o variaciones simples del mismo, pueden ayudar a resolver otros problemas de geometría computacional.

El algoritmo se ejecuta en  $O(n \lg n)$  tiempo, donde  $n$  es el número de segmentos que se nos dan. Determina únicamente si existe o no alguna intersección; no imprime todas las intersecciones. (Según el ejercicio 33.2-1, toma  $O(n^2)$  tiempo en el peor de los casos encontrar todas las intersecciones en un conjunto de  $n$  segmentos de línea).

En el barrido, una línea de barrido vertical imaginaria pasa a través del conjunto dado de objetos geométricos, generalmente de izquierda a derecha. Tratamos la dimensión espacial por la que se mueve la línea de barrido, en este caso la dimensión  $x$ , como una dimensión del tiempo. El barrido proporciona un método para ordenar objetos geométricos, generalmente colocándolos en una estructura de datos dinámica y para aprovechar las relaciones entre ellos. El algoritmo de intersección de segmento de línea de esta sección considera todos los puntos finales de segmento de línea en orden de izquierda a derecha y busca una intersección cada vez que encuentra un punto final.

Para describir y demostrar que nuestro algoritmo es correcto para determinar si dos cualesquiera de  $n$  segmentos de línea se intersecan, haremos dos suposiciones simplificadoras. Primero, asumimos que ningún segmento de entrada es vertical. En segundo lugar, asumimos que no hay tres segmentos de entrada que se crucen en un solo punto. Los ejercicios 33.2-8 y 33.2-9 le piden que demuestre que el algoritmo es lo suficientemente robusto como para que solo necesite una ligera modificación para funcionar incluso cuando estas suposiciones no se cumplen. De hecho, eliminar tales suposiciones simplificadoras y tratar con las condiciones de contorno a menudo presenta los desafíos más difíciles cuando se programan algoritmos de geometría computacional y se prueba su corrección.

#### Pedir segmentos

Debido a que asumimos que no hay segmentos verticales, sabemos que cualquier segmento de entrada que interseque una línea de barrido vertical dada, la interseca en un solo punto. Así, podemos ordenar los segmentos que intersecan una línea de barrido vertical según las coordenadas y de los puntos de intersección.

Para ser más precisos, considere dos segmentos  $s_1$  y  $s_2$ . Decimos que estos segmentos son comparables en  $x$  si la línea de barrido vertical con  $x$ -coordenada  $x$  los cruza a ambos. Decimos que  $s_1$  está por encima de  $s_2$  en  $x$ , escrito  $s_1 <_x s_2$ , si  $s_1$  y  $s_2$  son comparables en  $x$  y la intersección de  $s_1$  con la línea de barrido en  $x$  es más alta que la intersección de  $s_2$  con la misma línea de barrido, o si  $s_1$  y  $s_2$  se intersecan en la línea de barrido. En la figura 33.4(a), por ejemplo, tenemos las relaciones  $a <_r c$ ,  $a <_t b$ ,  $b <_t c$ ,  $a <_t c$  y  $b <_u c$ . El segmento  $d$  no es comparable con ningún otro segmento.

Para cualquier  $x$  dada, la relación “ $<_x$ ” es un preorden total (ver Sección B.2) para todos los segmentos que intersecan la línea de barrido en  $x$ . Es decir, la relación es transitiva, y si los segmentos  $s_1$  y  $s_2$  intersecan la línea de barrido en  $x$ , entonces  $s_1 <_x s_2$  o  $s_2 <_x s_1$ , o ambos (si  $s_1$  y  $s_2$  se intersecan en la línea de barrido). (La relación  $<_x$  es

## 33.2 Determinar si algún par de segmentos se interseca

1023

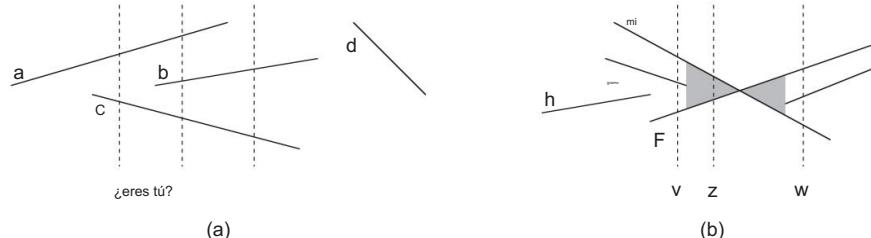


Figura 33.4 El orden entre los segmentos de línea en varias líneas de barrido vertical. (a) Tenemos  $a < r c$ ,  $a < t b$ ,  $b < t c$ ,  $a < t c$  y  $b < u c$ . El segmento  $d$  es comparable con ningún otro segmento mostrado. (b) Cuando los segmentos  $e y f$  se intersecan, invierten sus órdenes: tenemos  $e < f$  pero  $f < w e$ . Cualquier línea de barrido (como  $'$ ) que pasa por la región sombreada tiene  $e$  y  $f$  consecutivos en el orden dado por la relación  $<$ .

también reflexivo, pero ni simétrico ni antisimétrico.) El preorden total puede diferir para diferentes valores de  $x$ , sin embargo, cuando los segmentos entran y salen del orden.

Un segmento entra en la ordenación cuando el barrido encuentra su extremo izquierdo y abandona la ordenación cuando encuentra su extremo derecho.

¿Qué sucede cuando la línea de barrido pasa por la intersección de dos segmentos? Como muestra la figura 33.4(b), los segmentos invierten sus posiciones en el preorden total. Las líneas de barrido  $yw$  están a la izquierda y derecha, respectivamente, del punto de intersección de los segmentos  $e$  y  $f$ , y tenemos  $e < f$  y  $f < w e$ . Tenga en cuenta que debido a que asumimos que no hay tres segmentos que se intersequen en el mismo punto, debe haber alguna línea de barrido vertical  $x$  para la cual los segmentos de intersección  $e$  y  $f$  sean consecutivos en el preorden total  $<x$ . Cualquier línea de barrido que pase por la región sombreada de la figura 33.4(b), como  $'$ , tiene  $e$  y  $f$  consecutivos en su preorden total.

#### Mover la línea de barrido

Los algoritmos de barrido suelen gestionar dos conjuntos de datos:

1. El estado de la línea de barrido proporciona las relaciones entre los objetos que el barrido línea se cruza.
2. El horario de puntos de eventos es una secuencia de puntos, llamados puntos de eventos, que ordenamos de izquierda a derecha de acuerdo con sus coordenadas  $x$ . A medida que el barrido avanza de izquierda a derecha, cada vez que la línea de barrido alcanza la coordenada  $x$  de un punto de evento, el barrido se detiene, procesa el punto de evento y luego se reanuda. Los cambios en el estado de la línea de barrido ocurren solo en los puntos de evento.

Para algunos algoritmos (el algoritmo solicitado en el ejercicio 33.2-7, por ejemplo), el programa de punto de evento se desarrolla dinámicamente a medida que avanza el algoritmo. Sin embargo, el algoritmo en cuestión determina todos los puntos de evento antes del barrido, basándose en

únicamente en propiedades simples de los datos de entrada. En particular, cada punto final de segmento es un punto de evento. Ordenamos los puntos finales del segmento aumentando la coordenada  $x$  y procedemos de izquierda a derecha. (Si dos o más extremos son encubiertos, es decir, tienen la misma coordenada  $x$ , desempatamos colocando todos los extremos izquierdos encubiertos antes de los extremos derechos encubiertos. Dentro de un conjunto de extremos izquierdos encubiertos, colocamos aquellos con y inferior -coordenadas primero, y hacemos lo mismo dentro de un conjunto de extremos derechos de covrtical.) Cuando encontramos el extremo izquierdo de un segmento, insertamos el segmento en el estado de la línea de barrido, y eliminamos el segmento del estado de la línea de barrido en encontrando su punto final derecho. Cada vez que dos segmentos se vuelven consecutivos por primera vez en el pedido anticipado

El estado de la linea de barrido es un total de  $T$  operaciones total, verificamos si se cruzan. para lo cual requerimos lo siguiente de preorden:

INSERTAR.T;  $s/$ : insertar segmento  $s$  en  $T$

BORRAR.T;  $s/$ : borra el segmento  $s$  de  $T$

ABOVE.T;  $s/$ : devuelve el segmento inmediatamente superior al segmento  $s$

en  $T$  BELOW.T;  $s/$ : devuelve el segmento inmediatamente debajo del segmento  $s$  en  $T$

Es posible que los segmentos  $s_1$  y  $s_2$  estén uno encima del otro en el preorden total  $T$ ; esta situación puede ocurrir si  $s_1$  y  $s_2$  se cruzan en la línea de barrido cuyo preorden total está dado por  $T$ . En este caso, los dos segmentos pueden aparecer en cualquier orden en  $T$ . Si la entrada

contiene  $n$  segmentos, podemos realizar cada una de las operaciones INSERTAR , ELIMINAR, ARRIBA y ABAJO en tiempo  $O.\lg n /$  usando árboles rojo-negro. Recuerde que las operaciones de árbol rojo-negro del capítulo 13 implican la comparación de claves. Podemos reemplazar las comparaciones clave por comparaciones que usan productos cruzados para determinar el orden relativo de dos segmentos (vea el Ejercicio 33.2-2).

#### Pseudocódigo de intersección de segmento

El siguiente algoritmo toma como entrada un conjunto  $S$  de  $n$  segmentos de línea, devolviendo el valor booleano VERDADERO si cualquier par de segmentos en  $S$  se cruza, y FALSO en caso contrario. Un árbol rojo-negro mantiene el preorden total  $T$

ANY-SEGMENTS-INTERSECT.S /

1 TD; 2

ordenar los extremos de los segmentos en S de izquierda a derecha,  
 rompiendo empates poniendo los puntos finales izquierdos antes que los  
 puntos finales derechos y rompiendo más empates poniendo puntos  
 con coordenadas y  
 más bajas primero 3 para cada punto p en la lista ordenada de  
 puntos finales 4 si p es el punto final izquierdo de un  
 5                       segmento s  
 6                       INSERTAR.T; s/ si (ARRIBA.T; s/ existe y se cruza con s)  
                          o (ABAJO.T; s/ existe y se cruza con s)  
 7                       devolver VERDADERO  
 8                       si p es el extremo derecho de un segmento s  
 9                       si ambos ARRIBA.T; s/ y ABAJO.T; s/ existe  
                          y ARRIBA.T; s/ interseca ABAJO.T; s/ devuelve  
 10                      TRUE DELETE.T;  
 11                      s/ 12 devuelve  
 FALSO

La figura 33.5 ilustra cómo funciona el algoritmo. La línea 1 inicializa el pedido anticipado total para que esté vacío. La línea 2 determina el cronograma de puntos de eventos ordenando los puntos finales de  $2n$  segmentos de izquierda a derecha, rompiendo los empates como se describe arriba. Una forma de realizar la línea 2 es ordenar lexicográficamente los extremos en  $.x; mi; y/$ , donde  $x$  e  $y$  son las coordenadas habituales,  $e D 0$  para un extremo izquierdo y  $e D 1$  para un extremo derecho.

Cada iteración del bucle for de las líneas 3 a 11 procesa un punto de evento  $p$ . Si  $p$  es el extremo izquierdo de un segmento  $s$ , la línea 5 suma  $s$  al preorden total, y las líneas 6 y 7 devuelven TRUE si  $s$  se cruza con cualquiera de los segmentos con los que es consecutivo en el preorden total definido por la línea de barrido que pasa por  $p$ . (Se produce una condición de contorno si  $p$  se encuentra en otro segmento  $s_0$ . En este caso, solo requerimos que  $s$  y  $s_0$  se coloquen consecutivamente en  $T$ .) Si  $p$  es el extremo derecho de un segmento  $s$ , entonces necesitamos eliminar  $s$  del preorden total. Pero primero, las líneas 9 y 10 devuelven VERDADERO si hay una intersección entre los segmentos que rodean a  $s$  en el orden previo total definido por la línea de barrido que pasa por  $p$ . Si estos segmentos no se cruzan, la línea 11 elimina los segmentos del pedido anticipado total. Si los segmentos que rodean al segmento  $s$  se cruzan, se habrían convertido en consecutivos después de eliminar  $s$  si la declaración de retorno en la línea 10 no hubiera impedido que la línea 11 se ejecutara. El argumento de la corrección, que sigue, aclarará por qué es suficiente comprobar los segmentos que rodean a  $s$ . Finalmente, si nunca encontramos intersecciones después de haber procesado los  $2n$  puntos de evento, la línea 12 devuelve FALSO.

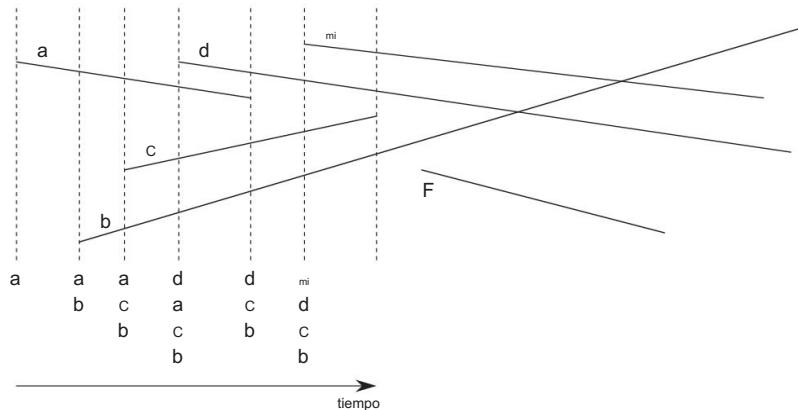


Figura 33.5 La ejecución de ANY-SEGMENTS-INTERSECT. Cada línea discontinua es la línea de barrido en un punto de evento. Excepto por la línea de barrido más a la derecha, el orden de los nombres de los segmentos debajo de cada línea de barrido corresponde al preorden total T al final del bucle for que procesa el punto de evento correspondiente. La línea de barrido más a la derecha se produce cuando se procesa el extremo derecho del segmento c; debido a que los segmentos d y b rodean c y se intersecan entre sí, el procedimiento devuelve VERDADERO.

### Exactitud

Para mostrar que ANY-SEGMENTS-INTERSECT es correcto, probaremos que la llamada ANY-SEGMENTS-INTERSECT.S / devuelve TRUE si y solo si hay una intersección entre los segmentos en S.

Es fácil ver que ANY-SEGMENTS-INTERSECT devuelve VERDADERO (en las líneas 7 y 10) solo si encuentra una intersección entre dos de los segmentos de entrada. Por lo tanto, si devuelve VERDADERO, hay una intersección.

También necesitamos mostrar lo contrario: que si hay una intersección, CUALQUIER SEGMENTO-INTERSECCIÓN devuelve VERDADERO. Supongamos que hay al menos una intersección. Sea p el punto de intersección más a la izquierda, rompiendo vínculos eligiendo el punto con la coordenada y más baja, y sean a y b los segmentos que se intersecan en p. Como no hay intersecciones a la izquierda de p, el orden dado por T es correcto en todos los puntos a la izquierda de p. Como no hay tres segmentos que se intersequen en el mismo punto, a y b se vuelven consecutivos en el preorden total en alguna línea de barrido.<sup>2</sup> Además, 'está a la izquierda de p o pasa por p. Algún extremo del segmento q en la línea de barrido '

---

<sup>2</sup>Si permitimos que tres segmentos se intersequen en el mismo punto, puede haber un segmento intermedio c que interseque tanto a como a b en el punto p. Es decir, podemos tener  $a < w c < w b$  para todas las líneas de barrido w a la izquierda de p para las cuales  $a < w b$ . El ejercicio 33.2-8 le pide que demuestre que CUALQUIER-SEGMENTO-INTERSECCIÓN es correcto incluso si tres segmentos se intersecan en el mismo punto.

es el punto de evento en el que a y b se vuelven consecutivos en el preorden total. Si p está en la línea de barrido ', entonces q D p. Si p no está en la línea de barrido ', entonces q está a la izquierda de p. En cualquier caso, la orden dada por T es correcta justo antes de encontrar q. (Aquí es donde usamos el orden lexicográfico en el que el algoritmo procesa los puntos de evento. Debido a que p es el más bajo de los puntos de intersección más a la izquierda, incluso si p está en la línea de barrido ' y algún otro punto de intersección p0 está en ', el punto de evento q D p se procesa antes que la otra intersección p0 puede interferir con el preorden total . T Además, incluso si p es el punto final izquierdo de un segmento, digamos a, y el punto final derecho del otro segmento, digamos b, porque los eventos del punto final izquierdo ocurren antes eventos del extremo derecho, el segmento b está en T al encontrar por primera vez el segmento a.) El punto de evento q es procesado por ANY-SEGMENTS-INTERSECT o no es procesado.

Si q es procesada por ANY-SEGMENTS-INTERSECT, solo dos acciones posibles puede ocurrir:

1. Se inserta a o b en T , y el otro segmento está por encima o por debajo de él en el pedido anticipado total. Las líneas 4 a 7 detectan este caso.
2. Los segmentos a y b ya están en T , y se elimina un segmento entre ellos en el preorden total, haciendo que a y b se vuelvan consecutivos. Las líneas 8 a 11 detectan esto caso.

En cualquier caso, encontramos la intersección p y ANY-SEGMENTS-INTERSECT devuelve VERDADERO.

Si el punto de evento q no es procesado por ANY-SEGMENTS-INTERSECT, el procedimiento debe haber regresado antes de procesar todos los puntos de evento. Esta situación podría haber ocurrido solo si ANY-SEGMENTS-INTERSECT ya hubiera encontrado una intersección y hubiera devuelto TRUE.

Por lo tanto, si hay una intersección, ANY-SEGMENTS-INTERSECT devuelve VERDADERO. Como ya hemos visto, si ANY-SEGMENTS-INTERSECT devuelve TRUE, hay una intersección. Por lo tanto, ANY-SEGMENTS-INTERSECT siempre devuelve una correcta respuesta.

#### Tiempo de ejecución

Si el conjunto S contiene n segmentos, ANY-SEGMENTS-INTERSECT se ejecuta en el tiempo en  $\lg n!$ . La línea 1 toma  $O.1/\lg n$  tiempo. La línea 2 toma  $O.\lg n/\lg n!$  tiempo, usando merge sort o heapsort. El ciclo for de las líneas 3 a 11 itera como máximo una vez por punto de evento y, por lo tanto, con  $2n$  puntos de evento, el ciclo itera como máximo  $2n$  veces. Cada iteración toma  $O.\lg n/\lg n!$  tiempo, ya que cada operación de árbol rojo-negro toma  $O.\lg n/\lg n!$  tiempo y, usando el método de la Sección 33.1, cada prueba de intersección toma  $O.1/\lg n!$  de tiempo. El tiempo total es pues  $O.\lg n/\lg n!$ .

**Ejercicios****33.2-1**

Muestre que un conjunto de  $n$  segmentos de línea puede contener  $\binom{n}{2}$  intersecciones.

**33.2-2**

Dados dos segmentos  $a$  y  $b$  que son comparables en  $x$ , muestre cómo determinar en  $O(1)$  tiempo cuál de  $a <_x b$  o  $b <_x a$  se cumple. Suponga que ninguno de los segmentos es vertical. (Sugerencia: si  $a$  y  $b$  no se intersecan, solo puede usar productos cruzados).

Si  $a$  y  $b$  se intersecan, lo cual, por supuesto, puede determinar usando solo productos cruzados, aún puede usar solo sumas, restas y multiplicaciones, evitando la división. Por supuesto, en la aplicación de la relación  $<_x$  utilizada aquí, si  $a$  y  $b$  se intersecan, podemos detenernos y declarar que hemos encontrado una intersección).

**33.2-3**

El profesor Mason sugiere que modifiquemos ANY-SEGMENTS-INTERSECT para que, en lugar de regresar al encontrar una intersección, imprima los segmentos que se intersecan y continúe con la siguiente iteración del bucle for . El profesor llama al procedimiento resultante PRINT-INTERSECTING-SEGMENTS y afirma que imprime todas las intersecciones, de izquierda a derecha, tal como ocurren en el conjunto de segmentos de línea. El profesor Dixon no está de acuerdo y afirma que la idea del profesor Mason es incorrecta. ¿Qué profesor tiene razón? ¿PRINT-INTERSECTING-SEGMENTS siempre encontrará primero la intersección más a la izquierda? ¿Encontrará siempre todas las intersecciones?

**33.2-4**

Proporcione un algoritmo On lg  $n$ -time para determinar si un polígono de  $n$  vértices es simple.

**33.2-5**

Proporcione un algoritmo On lg  $n$ -time para determinar si dos polígonos simples con un total de  $n$  vértices se intersecan.

**33.2-6**

Un disco consta de un círculo más su interior y se representa por su punto central y su radio. Dos discos se intersecan si tienen algún punto en común. Proporcione un algoritmo On lg  $n$ - time para determinar si dos discos en un conjunto de  $n$  se cruzan.

**33.2-7**

Dado un conjunto de  $n$  segmentos de línea que contienen un total de  $k$  intersecciones, muestre cómo generar todas las  $k$  intersecciones en  $O(n C k) / \lg n$  tiempo.

## 33.2-8

Argumente que ANY-SEGMENTS-INTERSECT funciona correctamente incluso si tres o más segmentos se intersecan en el mismo punto.

## 33.2-9

Muestre que ANY-SEGMENTS-INTERSECT funciona correctamente en presencia de segmentos verticales si tratamos el extremo inferior de un segmento vertical como si fuera un extremo izquierdo y el extremo superior como si fuera un extremo derecho. ¿Cómo cambia su respuesta al ejercicio 33.2-2 si permitimos segmentos verticales?

### 33.3 Encontrar el casco convexo

El casco convexo de un conjunto Q de puntos, denotado por CH.Q/, es el polígono convexo P más pequeño para el cual cada punto en Q está en el límite de P o en su interior. (Consulte el ejercicio 33.1-5 para obtener una definición precisa de un polígono convexo). Suponemos implícitamente que todos los puntos del conjunto Q son únicos y que Q contiene al menos tres puntos que no son colineales. Intuitivamente, podemos pensar en cada punto de Q como si fuera un clavo que sobresale de una tabla. El casco convexo es entonces la forma formada por una banda de goma apretada que rodea todos los clavos. La figura 33.6 muestra un conjunto de puntos y su casco convexo.

En esta sección, presentaremos dos algoritmos que calculan la envolvente convexa de un conjunto de n puntos. Ambos algoritmos generan los vértices del casco convexo en el sentido contrario a las agujas del reloj. El primero, conocido como exploración de Graham, se ejecuta en  $O(n \lg n)$  time. La segunda, llamada marcha de Jarvis, transcurre en tiempo  $O(nh)$ , donde h es el número de vértices del casco convexo. Como ilustra la figura 33.6, cada vértice de CH.Q/ es un

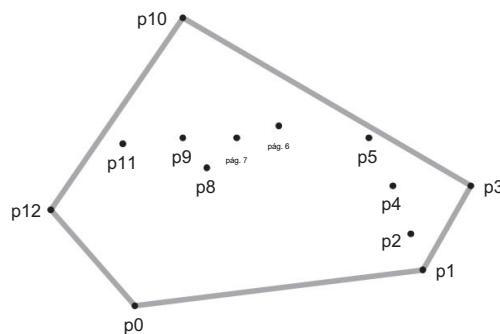


Figura 33.6 Un conjunto de puntos QD fp0; p1;:::;p12g con su casco convexo CH.Q/ en gris.

punto en Q. Ambos algoritmos explotan esta propiedad, decidiendo qué vértices en Q mantener como vértices del casco convexo y qué vértices en Q rechazar.

Podemos calcular casclos convexos en  $O(n \lg n)$  time por cualquiera de varios métodos. Tanto el escaneo de Graham como la marcha de Jarvis usan una técnica llamada "barrido rotacional", que procesa los vértices en el orden de los ángulos polares que forman con un vértice de referencia. Otros métodos incluyen los siguientes:

En el método incremental, primero ordenamos los puntos de izquierda a derecha, dando como resultado una secuencia  $hp_1; p_2; \dots; p_n$ . En la  $i$ -ésima etapa, actualizamos el casco convexo de los  $i$  puntos más a la izquierda,  $CH.fp_1; p_2; \dots; p_{i-1}g$ , según el  $i$ -ésimo punto de  $i$  a la izquierda, formando así  $CH.fp_1; p_2; \dots; p_i g$ . El ejercicio 33.3-6 le pregunta cómo implementar este método para obtener un total de  $O(n \lg n)$  time.

En el método divide y vencerás, dividimos el conjunto de  $n$  puntos en  $\sqrt{n}$  tiempo en dos subconjuntos, uno que contiene los puntos  $d_n=2e$  más a la izquierda y otro que contiene los puntos  $b_n=2c$  más a la derecha, calculamos recursivamente las envolventes convexas de los subconjuntos, y luego, por medio de un método inteligente, combinamos los casclos en  $O(n)$  time. El tiempo de ejecución se describe mediante la recurrencia familiar  $T(n) = 2T(\sqrt{n}) + O(n)$ , por lo que el método divide y vencerás se ejecuta en  $O(n \lg n)$  time.

El método de podar y buscar es similar al algoritmo de mediana de tiempo lineal del peor de los casos de la Sección 9.3. Con este método, encontramos la parte superior (o "cadena superior") del casco convexo descartando repetidamente una fracción constante de los puntos restantes hasta que solo quede la cadena superior del casco convexo. Luego hacemos lo mismo para la cadena inferior. Este método es asintóticamente el más rápido: si el casco convexo contiene  $h$  vértices, se ejecuta solo en  $O(h \lg h)$  time.

Calcular la envolvente convexa de un conjunto de puntos es un problema interesante por derecho propio. Además, los algoritmos para algunos otros problemas de geometría computacional comienzan calculando un casco convexo. Considere, por ejemplo, el problema bidimensional del par más lejano: tenemos un conjunto de  $n$  puntos en el plano y deseamos encontrar los dos puntos cuya distancia entre sí sea máxima. Como el Ejercicio 33.3-3 le pide que demuestre, estos dos puntos deben ser vértices del casco convexo. Aunque no lo probaremos aquí, podemos encontrar el par de vértices más alejados de un polígono convexo de  $n$  vértices en  $O(n^2)$  time. Por lo tanto, calculando la envolvente convexa de los  $n$  puntos de entrada en  $O(n \lg n)$  time y luego encontrando el par más lejano de los vértices resultantes del polígono convexo, podemos encontrar el par de puntos más lejano en cualquier conjunto de  $n$  puntos en  $O(n \lg n)$  time.

#### escaneo de graham

El escaneo de Graham resuelve el problema del casco convexo al mantener una pila  $S$  de puntos candidatos. Empuja cada punto del conjunto de entrada  $Q$  a la pila una vez,

y finalmente saca de la pila cada punto que no es un vértice de CH.Q/.

Cuando el algoritmo termina, la pila S contiene exactamente los vértices de CH.Q/, en el orden contrario a las agujas del reloj de su aparición en el límite.

El procedimiento GRAHAM-SCAN toma como entrada un conjunto Q de puntos, donde  $jQj \geq 3$ . Llama a las funciones TOP.S /, que devuelve el punto en la parte superior de la pila S sin cambiar S, y NEXT-TO-TOP.S /, que devuelve el punto una entrada por debajo de la parte superior de la pila S sin cambiar S. Como probaré en un momento, la pila S devuelta por GRAHAM-SCAN contiene, de abajo hacia arriba, exactamente los vértices de CH.Q/ en sentido antihorario.

#### GRAHAM-SCAN.Q/ 1

```

sea p0 el punto en Q con la coordenada y mínima,
o el punto más a la izquierda en caso de
empate 2 let hp1; p2;:::;pmi ser los puntos restantes en Q,
ordenados por ángulo polar en orden antihorario alrededor de p0
(si más de un punto tiene el mismo ángulo, elimine todos menos el
que está más alejado de p0) 3 sea
S una pila vacía 4 PUSH.p0;S/
5 PUSH.p1; S/ 6
EMPUJAR.p2;S/

```

7 para i D 3 a m

```

8     mientras que el ángulo formado por los puntos NEXT-TO-TOP.S /,
          TOP.S / y pi hace un giro no a
9         la
           izquierda
POP.S / PUSH.pi;S/ 10 11 return S

```

La Figura 33.7 ilustra el progreso de GRAHAM-SCAN. La línea 1 elige el punto  $p_0$  como el punto con la coordenada y más baja, eligiendo el punto más a la izquierda en caso de empate. Como no hay ningún punto en Q que esté debajo de  $p_0$  y cualquier otro punto con la misma coordenada y está a su derecha,  $p_0$  debe ser un vértice de CH.Q/. La línea 2 ordena los puntos restantes de Q por ángulo polar con respecto a  $p_0$ , utilizando el mismo método (comparación de productos cruzados) que en el ejercicio 33.1-3. Si dos o más puntos tienen el mismo ángulo polar con respecto a  $p_0$ , todos menos el punto más lejano son combinaciones convexas de  $p_0$  y el punto más lejano, por lo que los eliminamos por completo de nuestra consideración.

Dejamos que  $m$  denote el número de puntos distintos de  $p_0$  que quedan.

El ángulo polar, medido en radianes, de cada punto de Q relativo a  $p_0$  está en el intervalo semiabierto  $\text{CE}0; \pi/$ . Dado que los puntos se ordenan según los ángulos polares, se ordenan en sentido contrario a las agujas del reloj con respecto a  $p_0$ . Designamos esta secuencia ordenada de puntos por  $hp1; p2;:::;pmi$ . Tenga en cuenta que los puntos  $p_1$  y  $p_m$  son vértices

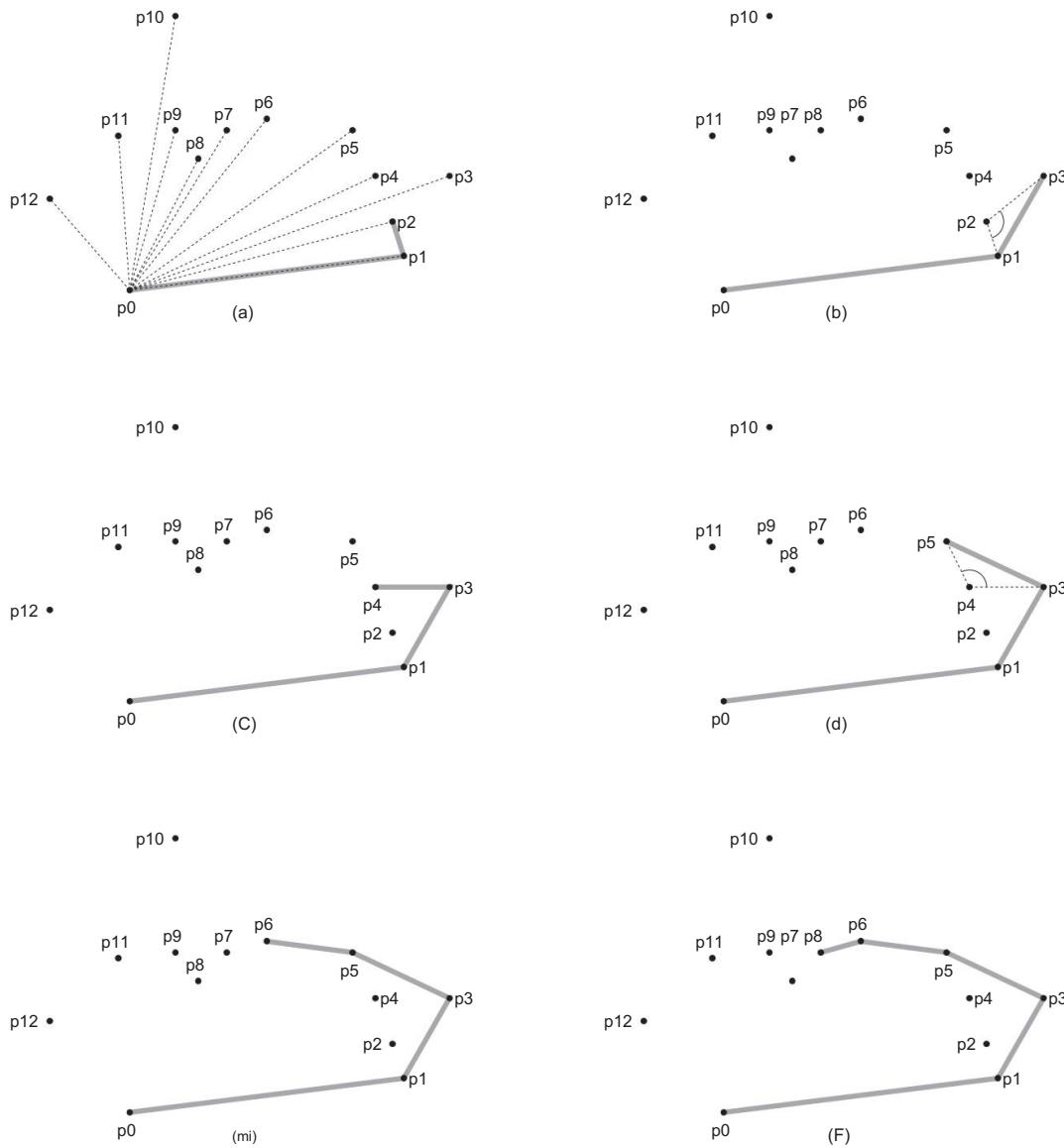


Figura 33.7 La ejecución de GRAHAM-SCAN en el conjunto Q de la Figura 33.6. El casco convexo actual contenido en la pila S se muestra en gris en cada paso. (a) La secuencia  $hp_1; p_2; \dots; p_{12}i$  de puntos numerados en orden de ángulo polar creciente en relación con  $p_0$ , y la pila inicial S que contiene  $p_0, p_1$  y  $p_2$ . (b)–(k) Apila S después de cada iteración del ciclo for de las líneas 7–10. Las líneas discontinuas muestran giros que no son a la izquierda, lo que hace que los puntos se extraigan de la pila. En la parte (h), por ejemplo, el giro a la derecha en el ángulo  $\hat{p}_7p_8p_9$  hace que se levante  $p_8$ , y luego el giro a la derecha en el ángulo  $\hat{p}_6p_7p_9$  hace que se levante  $p_7$ .

## 33.3 Encontrar el casco convexo

1033

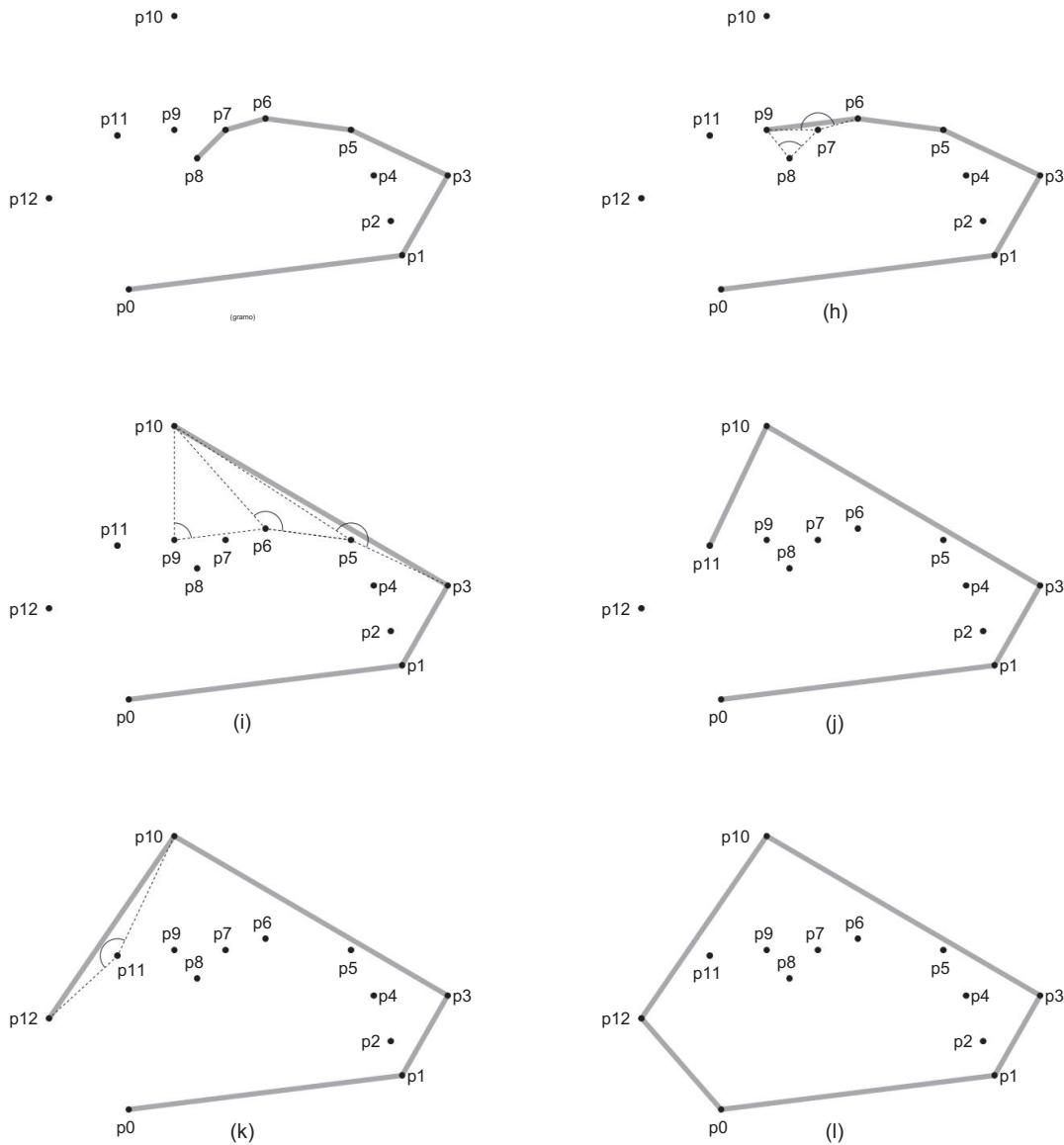


Figura 33.7, continuación (l) El casco convexo devuelto por el procedimiento, que coincide con el de la Figura 33.6.

de CH.Q/ (ver Ejercicio 33.3-1). La figura 33.7(a) muestra los puntos de la figura 33.6 numerados secuencialmente en orden de ángulo polar creciente en relación con p0.

El resto del procedimiento utiliza la pila S. Las líneas 3 a 6 inicializan la pila para que contenga, de abajo hacia arriba, los tres primeros puntos p0, p1 y p2. La figura 33.7(a) muestra la pila inicial S. El ciclo for de las líneas 7 a 10 itera una vez para cada punto en la subsecuencia hp3; p4;:::;pmi. Veremos que después de procesar el punto pi, la pila S contiene, de abajo hacia arriba, los vértices de CH.fp0; p1;:::;pi g/ en sentido antihorario. El ciclo while de las líneas 8–9 elimina puntos de la pila si encontramos que no son vértices del casco convexo. Cuando atravesamos el casco convexo en sentido contrario a las agujas del reloj, debemos girar a la izquierda en cada vértice. Por lo tanto, cada vez que el bucle while encuentra un vértice en el que hacemos un giro que no sea a la izquierda, extraemos el vértice de la pila. (Al verificar si hay un giro que no sea a la izquierda, en lugar de solo un giro a la derecha, esta prueba excluye la posibilidad de un ángulo recto en un vértice del casco convexo resultante. No queremos ángulos rectos, ya que ningún vértice de un polígono convexo puede ser un ángulo convexo). combinación de otros vértices del polígono.) Después de extraer todos los vértices que no giran a la izquierda cuando se dirigen hacia el punto pi , colocamos pi en la pila.

Las figuras 33.7(b)–(k) muestran el estado de la pila S después de cada iteración del bucle for . Finalmente, GRAHAM-SCAN devuelve la pila S en la línea 11. La figura 33.7(l) muestra el casco convexo correspondiente.

El siguiente teorema prueba formalmente la corrección de GRAHAM-SCAN.

#### Teorema 33.1 (Corrección del escaneo de Graham)

Si GRAHAM-SCAN se ejecuta en un conjunto Q de puntos, donde jQj 3, entonces, al terminar, la pila S consta, de abajo hacia arriba, exactamente de los vértices de CH.Q/ en sentido antihorario.

Prueba Despues de la línea 2, tenemos la sucesión de puntos hp1; p2; ::; pm. Definamos, para i D 2; 3; : : : ; m, el subconjunto de puntos Qi D fp0; p1;:::;pi g. Los puntos en Q Qm son los que se quitaron porque tenían el mismo ángulo polar relativo a p0 que algún punto en Qm; estos puntos no están en CH.Q/, por lo que CH.Qm/ D CH.Q/. Por lo tanto, basta con mostrar que cuando GRAHAM-SCAN termina, la pila S consta de los vértices de CH.Qm/ en el sentido contrario a las agujas del reloj, cuando se enumeran de abajo hacia arriba. Tenga en cuenta que así como p0, p1 y pm son vértices de CH.Q/, los puntos p0, p1 y pi son todos vértices de CH.Qi/.

La demostración utiliza el siguiente bucle invariante:

Al comienzo de cada iteración del ciclo for de las líneas 7 a 10, la pila S consta, de abajo hacia arriba, exactamente de los vértices de CH.Qi1/ en sentido antihorario.

Inicialización: El invariante se cumple la primera vez que ejecutamos la línea 7, ya que en ese momento la pila S consta exactamente de los vértices de Q2 D Qi1, y este conjunto de tres

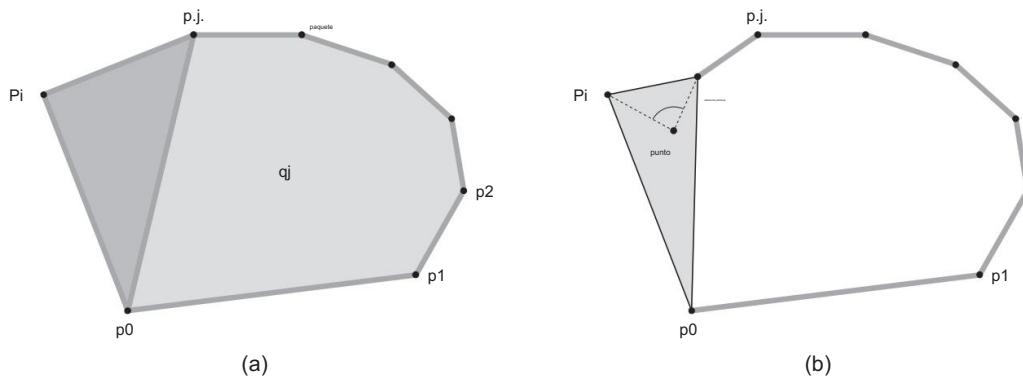


Figura 33.8 La prueba de corrección de GRAHAM-SCAN. (a) Debido a que el ángulo polar de  $\pi$  relativo a  $p_0$  es mayor que el ángulo polar de  $p_j$ , y debido a que el ángulo  $\angle p_k p_j \pi$  gira a la izquierda, sumando  $\pi$  a  $CH.Q_j /$  se obtienen exactamente los vértices de  $CH.Q_j / [fpi g/]$ . (b) Si el ángulo  $\angle p_r p_t \pi$  no gira a la izquierda, entonces  $p_t$  está en el interior del triángulo formado por  $p_0$ ,  $p_r$  y  $\pi$  o en un lado del triángulo, lo que significa que no puede ser un vértice de  $CH.Q_i /$ .

vértices forma su propio casco convexo. Además, aparecen en sentido contrario a las agujas del reloj de abajo hacia arriba.

Mantenimiento: al ingresar a una iteración del bucle `for`, el punto superior en la pila S es  $p_{i+1}$ , que se empujó al final de la iteración anterior (o antes de la primera iteración, cuando  $i = D - 3$ ). Sea  $p_j$  el punto superior de S después de ejecutar el bucle `while` de las líneas 8 y 9 pero antes de que la línea 10 presione  $\pi$ , y sea  $p_k$  el punto justo debajo de  $p_j$  en S. En el momento en que  $p_j$  es el punto superior de S y aún no ha presionado  $\pi$ , la pila S contiene exactamente los mismos puntos que contenía después de la iteración  $j$  del bucle `for`. Por el bucle invariante, por lo tanto, S contiene exactamente los vértices de  $CH.Q_j /$  en ese momento, y aparecen en orden antihorario de abajo hacia arriba.

Sigamos enfocándonos en este momento justo antes de presionar  $\pi$ . Sabemos que el ángulo polar de  $\pi$  con respecto a  $p_0$  es mayor que el ángulo polar de  $p_j$  y que el ángulo  $\angle p_k p_j \pi$  gira a la izquierda (de lo contrario, habríamos saltado  $p_j$ ).

Por lo tanto, debido a que S contiene exactamente los vértices de  $CH.Q_j /$ , vemos en la figura 33.8(a) que una vez que presionamos  $\pi$ , la pila S contendrá exactamente los vértices de  $CH.Q_j / [fpi g/]$ , aún en orden antihorario desde abajo hasta arriba.

Ahora mostramos que  $CH.Q_j / [fpi g/]$  es el mismo conjunto de puntos que  $CH.Q_i /$ . Considere cualquier punto  $p_t$  que se extraiga durante la iteración  $i$  del ciclo `for`, y sea  $p_r$  el punto justo debajo de  $p_t$  en la pila S en el momento en que se extraiga  $p_t$  ( $p_r$  podría ser  $p_j$ ).

El ángulo  $\angle p_r p_t \pi$  hace un giro no a la izquierda y el ángulo polar de  $p_t$  relativo a  $p_0$  es mayor que el ángulo polar de  $p_r$ .

Como muestra la figura 33.8(b),  $p_t$  debe

estar en el interior del triángulo formado por  $p_0$ ,  $p_r$  y  $p_i$  o en un lado de este triángulo (pero no es un vértice del triángulo). Claramente, dado que  $p_t$  está dentro de un triángulo formado por otros tres puntos de  $Q_i$ , no puede ser un vértice de  $CH.Q_i$ .

Como  $p_t$  no es un vértice de  $CH.Q_i$ , tenemos que

$$CH.Q_i fpt g / D CH.Q_i : \quad (33.1)$$

Sea  $P_i$  el conjunto de puntos que se trajeron durante la iteración  $i$  del bucle for.

Dado que la igualdad (33.1) se aplica a todos los puntos de  $P_i$ , podemos aplicarla repetidamente para mostrar que  $CH.Q_i P_i / D CH.Q_i$ . Pero  $Q_i P_i D Q_j [ fpi g$ , por lo que concluimos que  $CH.Q_j [ fpi g / D CH.Q_i P_i / D CH.Q_i$ .

Hemos demostrado que una vez que presionamos  $p_i$ , la pila  $S$  contiene exactamente los vértices de  $CH.Q_i$  en sentido antihorario de abajo hacia arriba. Incrementar  $i$  hará que el ciclo invariante se mantenga para la siguiente iteración.

Terminación: cuando el bucle termina, tenemos  $i \leq m - 1$ , por lo que la invariante del bucle implica que la pila  $S$  consta exactamente de los vértices de  $CH.Q_m$ , que es  $CH.Q$ , en sentido contrario a las agujas del reloj de abajo hacia arriba. Esto completa la prueba. ■

Ahora mostramos que el tiempo de ejecución de GRAHAM-SCAN es  $O(n \lg n)$ , donde  $n = |Q|$ . La línea 1 toma  $O(n)$  tiempo. La línea 2 toma  $O(n \lg n)$  tiempo, usando merge sort o heapsort para ordenar los ángulos polares y el método de productos cruzados de la Sección 33.1 para comparar ángulos. (Podemos eliminar todos menos el punto más lejano con el mismo ángulo polar en total de  $O(n \lg n)$  tiempo sobre todos los  $n$  puntos). Las líneas 3–6 toman  $O(1)$  tiempo. Debido a que  $m \geq 1$ , el bucle for de las líneas 7 a 10 se ejecuta como máximo  $n - 3$  veces. Dado que PUSH toma  $O(1)$  tiempo, cada iteración toma  $O(1)$  tiempo excluyendo el tiempo empleado en el ciclo while de las líneas 8–9 y, por lo tanto, en general, el ciclo for toma  $O(n)$  tiempo excluyendo el ciclo while anidado.

Usamos un análisis agregado para mostrar que el ciclo while toma el tiempo de encendido/en general. Para  $i = 0; i < m; i++$ , empujamos cada punto  $p_i$  a la pila  $S$  exactamente una vez. Al igual que en el análisis del procedimiento MULTIPOL de la Sección 17.1, observamos que podemos extraer como máximo el número de elementos que empujamos. Al menos tres puntos ( $p_0, p_1$  y  $p_m$ ) nunca se extraen de la pila, de modo que, de hecho, se realizan como máximo  $m - 2$  operaciones POP en total. Cada iteración del bucle while realiza un POP, por lo que hay como máximo  $m - 2$  iteraciones del bucle while en total. Dado que la prueba en la línea 8 toma  $O(1)$  tiempo, cada llamada de POP toma  $O(1)$  tiempo, y  $m \geq 1$ , el tiempo total que toma el ciclo while es  $O(n)$ . Por lo tanto, el tiempo de ejecución de GRAHAM-SCAN es  $O(n \lg n)$ .

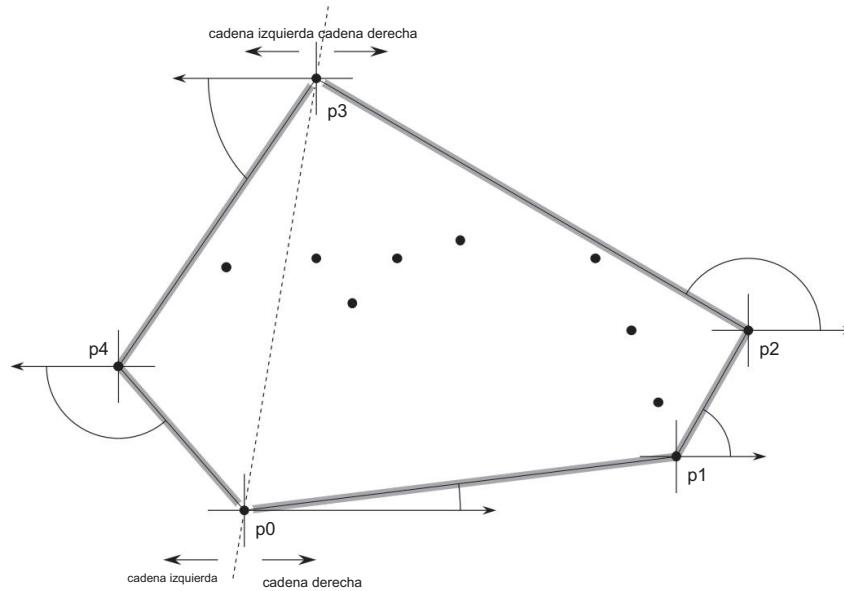


Figura 33.9 La operación de la marcha de Jarvis. Elegimos el primer vértice como el punto más bajo  $p_0$ . El siguiente vértice,  $p_1$ , tiene el ángulo polar más pequeño de cualquier punto con respecto a  $p_0$ . Entonces,  $p_2$  tiene el ángulo polar más pequeño con respecto a  $p_1$ . La cadena derecha llega hasta el punto más alto  $p_3$ . Luego, construimos la cadena izquierda encontrando los ángulos polares más pequeños con respecto al eje x negativo.

### la marcha de jarvis

La marcha de Jarvis calcula el casco convexo de un conjunto  $Q$  de puntos mediante una técnica conocida como envoltorio de paquete (o envoltorio de regalo). El algoritmo se ejecuta en el tiempo  $O.nh/$ , donde  $h$  es el número de vértices de  $CH.Q/$ . Cuando  $h$  es  $o.\lg n/$ , la marcha de Jarvis es asintóticamente más rápida que la exploración de Graham.

Intuitivamente, la marcha de Jarvis simula envolver un trozo de papel tenso alrededor del conjunto  $Q$ . Comenzamos pegando el extremo del papel en el punto más bajo del conjunto, es decir, en el mismo punto  $p_0$  con el que comenzamos el escaneo de Graham . Sabemos que este punto debe ser un vértice del casco convexo. Tiramos del papel hacia la derecha para que quede tenso y luego lo tiramos más arriba hasta que toque un punto. Este punto también debe ser un vértice del casco convexo. Manteniendo el papel tenso, continuamos de esta manera alrededor del conjunto de vértices hasta volver a nuestro punto original  $p_0$ .

Más formalmente, la marcha de Jarvis construye una secuencia  $HD hp_0; p_1; \dots; p_{h-1}$  de los vértices de  $CH.Q/$ . Empezamos con  $p_0$ . Como muestra la figura 33.9, el siguiente vértice  $p_1$  en el casco convexo tiene el ángulo polar más pequeño con respecto a  $p_0$ . (En caso de empate, elegimos el punto más alejado de  $p_0$ ). De manera similar,  $p_2$  tiene el ángulo polar más pequeño

con respecto a  $p_1$ , y así sucesivamente. Cuando alcanzamos el vértice más alto, digamos  $p_k$  (rompiendo vínculos eligiendo el vértice más lejano), hemos construido, como muestra la figura 33.9, la cadena derecha de  $CH.Q/$ . Para construir la cadena izquierda, comenzamos en  $p_k$  y elegimos  $p_{k-1}$  como el punto con el ángulo polar más pequeño con respecto a  $p_k$ , pero desde el eje  $x$  negativo. Continuamos formando la cadena izquierda tomando ángulos polares del eje  $x$  negativo, hasta que volvemos a nuestro vértice original  $p_0$ .

Podríamos implementar la marcha de Jarvis en un barrio conceptual alrededor del casco convexo, es decir, sin construir por separado las cadenas derecha e izquierda. Dichas implementaciones suelen hacer un seguimiento del ángulo del último lado del casco convexo elegido y requieren que la secuencia de ángulos de los lados del casco sea estrictamente creciente (en el rango de 0 a 2 radianes). La ventaja de construir cadenas separadas es que no necesitamos calcular ángulos explícitamente; las técnicas de la Sección 33.1 son suficientes para comparar ángulos.

Si se implementa correctamente, la marcha de Jarvis tiene un tiempo de ejecución de  $O.nh/$ . Para cada uno de los  $h$  vértices de  $CH.Q/$ , encontramos el vértice con el ángulo polar mínimo. Cada comparación entre ángulos polares toma  $O.1/$  de tiempo, usando las técnicas de la Sección 33.1. Como muestra la Sección 9.1, podemos calcular el mínimo de  $n$  valores en  $O.n/time$  si cada comparación toma  $O.1/time$ . Así, la marcha de Jarvis toma  $O.nh/time$ .

### Ejercicios

#### 33.3-1

Demuestre que en el procedimiento GRAHAM-SCAN, los puntos  $p_1$  y  $p_m$  deben ser vértices de  $CH.Q/$ .

#### 33.3-2

Considere un modelo de computación que admita sumas, comparaciones y multiplicaciones y para el cual existe un límite inferior de  $.n \lg n/$  para clasificar  $n$  números. Demuestre que  $.n \lg n/$  es una cota inferior para calcular, en orden, los vértices de la envolvente convexa de un conjunto de  $n$  puntos en tal modelo.

#### 33.3-3

Dado un conjunto de puntos  $Q$ , demuestre que el par de puntos más alejados entre sí deben ser vértices de  $CH.Q/$ .

#### 33.3-4

Para un polígono  $P$  dado y un punto  $q$  en su límite, la sombra de  $q$  es el conjunto de puntos  $r$  tales que el segmento  $qr$  está completamente en el límite o en el interior de  $P$ . Como ilustra la figura 33.10, un polígono  $P$  tiene forma de estrella si existe un punto  $p$  en el interior de  $P$  que está en la sombra de cada punto en el límite de  $P$ . El conjunto de todos esos puntos  $p$  se llama núcleo de  $P$ . Dado un  $n$ -vértice,

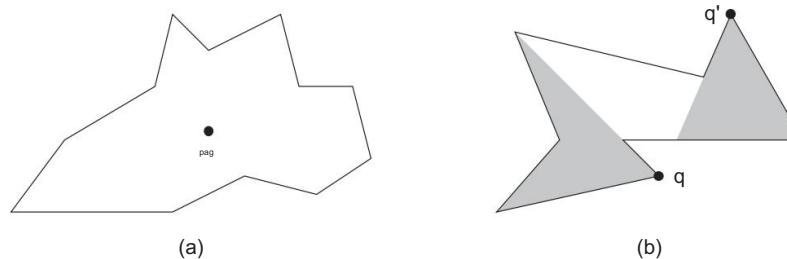


Figura 33.10 La definición de un polígono en forma de estrella, para usar en el Ejercicio 33.3-4. (a) Un polígono en forma de estrella. El segmento desde el punto  $p$  hasta cualquier punto  $q$  en el límite interseca al límite solo en  $q$ . (b) Un polígono sin forma de estrella. La región sombreada de la izquierda es la sombra de  $q$ , y la región sombreada de la derecha es la sombra de  $q_0$ . Dado que estas regiones están separadas, el núcleo está vacío.

polígono en forma de estrella  $P$  especificado por sus vértices en sentido antihorario, muestra cómo calcular CH.P / en On/ time.

### 33.3-5

En el problema de casco convexo en línea, se nos da el conjunto  $Q$  de  $n$  puntos, un punto a la vez. Después de recibir cada punto, calculamos la envolvente convexa de los puntos vistos hasta ahora. Obviamente, podríamos ejecutar el escaneo de Graham una vez para cada punto, con un tiempo de ejecución total de  $O.n^2 \lg n$ . Muestre cómo resolver el problema del casco convexo en línea en un tiempo total de  $O.n^2$ .

### 33.3-6 ?

Muestre cómo implementar el método incremental para calcular la envolvente convexa de  $n$  puntos para que se ejecute en On  $\lg n$  / time.

## 33.4 Encontrar el par de puntos más cercano

Ahora consideraremos el problema de encontrar el par de puntos más cercano en un conjunto  $Q$  de  $n$  2 puntos. "Más cercano" se refiere a la distancia euclídea habitual: la distancia entre los puntos  $p_1 D .x_1; y_1 / y p_2 D .x_2; y_2 /$  es  $p_1.x_1 x_2/2 C .y_1 y_2/2$ . Dos puntos del conjunto  $Q$  pueden ser coincidentes, en cuyo caso la distancia entre ellos es cero. Este problema tiene aplicaciones, por ejemplo, en sistemas de control de tráfico. Un sistema para controlar el tráfico aéreo o marítimo podría necesitar identificar los dos vehículos más cercanos para detectar posibles colisiones.

algoritmo de par más cercano de fuerza bruta simplemente analiza  $\frac{n(n-1)}{2}$  pares Un todos los puntos. En esta sección, describiremos un algoritmo divide y vencerás para

este problema, cuyo tiempo de ejecución se describe mediante la conocida recurrencia  $T(n) = 2T(n/2) + O(n)$ . Por lo tanto, este algoritmo usa solo  $O(n \lg n)$  time.

### El algoritmo divide y vencerás

Cada invocación recursiva del algoritmo toma como entrada un subconjunto PQ y matrices X e Y, cada una de las cuales contiene todos los puntos del subconjunto de entrada P. Los puntos en la matriz X se ordenan de modo que sus coordenadas x aumenten monótonamente. De manera similar, la matriz Y se ordena por coordenada y que aumenta monótonamente. Tenga en cuenta que para alcanzar el límite de tiempo de  $O(n \lg n)$ , no podemos darnos el lujo de ordenar en cada llamada recursiva; si lo hicieramos, la recurrencia para el tiempo de ejecución sería  $T(n) = 2T(n/2) + O(n \lg n)$ , cuya solución es  $T(n) = O(n \lg^2 n)$ . (Use la versión del método maestro dada en el ejercicio 4.6-2.) Veremos un poco más adelante cómo usar la "clasificación previa" para mantener esta propiedad ordenada sin clasificar en cada llamada recursiva.

Una invocación recursiva determinada con entradas P, X e Y primero verifica si  $jPj \leq 3$ . Si es así, la invocación simplemente realiza el método de fuerza bruta descrito anteriormente: prueba todos los pares de puntos  $(j, k)$  y devuelve el par más cercano. Si  $jPj > 3$ , la invocación recursiva lleva a cabo el paradigma divide y vencerás de la siguiente manera.

**Dividir:** Encuentre una línea vertical  $I$  que biseque el conjunto de puntos P en dos conjuntos  $PL$  y  $PR$  tales que  $|PL| \leq jPj/2$ ,  $|PR| \leq jPj/2$ , todos los puntos en  $PL$  están sobre o a la izquierda de la línea  $I$ , y todos los puntos en  $PR$  están sobre o a la derecha de  $I$ . Divida la matriz X en matrices  $XL$  y  $XR$ , que contienen los puntos de  $PL$  y  $PR$  respectivamente, ordenados por coordenada x creciente monótonamente. De manera similar, divida la matriz Y en matrices  $YL$  e  $YR$ , que contienen los puntos de  $PL$  y  $PR$  respectivamente, ordenados por coordenada y que aumenta monótonamente.

**Conquistar:** Habiendo dividido P en  $PL$  y  $PR$ , haga dos llamadas recursivas, una para encontrar el par de puntos más cercano en  $PL$  y la otra para encontrar el par de puntos más cercano en  $PR$ . Las entradas de la primera llamada son el subconjunto  $PL$  y las matrices  $XL$  e  $YL$ ; la segunda llamada recibe las entradas  $PR$ ,  $XR$  e  $YR$ . Sean  $iL$  e  $iR$ , respectivamente, las distancias del par más cercano devueltas para  $PL$  y  $PR$ , y sea  $iD = \min(iL, iR)$ .

**Combine:** El par más cercano es el par con distancia  $iD$  encontrado por una de las llamadas recursivas, o es un par de puntos con un punto en  $PL$  y el otro en  $PR$ .

El algoritmo determina si existe un par con un punto en  $PL$  y otro en  $PR$  y cuya distancia es menor que  $iD$ . Observa que si un par de puntos tiene una distancia menor que  $iD$ , ambos puntos del par deben estar dentro de  $iD$  unidades de la línea  $I$ . Por lo tanto, como muestra la figura 33.11(a), ambos deben residir en la franja vertical de  $2iD$  de ancho centrada en la línea  $I$ . Para encontrar tal par, si existe, hacemos lo siguiente:

1. Crear una matriz Y tira  $i^0$ , que es la matriz Y con todos los puntos que no están en el  $2i$  de ancho vertical eliminada. La matriz  $Y^{i^0}$  se ordena por coordenada y, al igual que  $Y_{i^0}$ .
2. Para cada punto p en la matriz  $Y^{i^0}$  intente encontrar puntos en  $Y^{i^0}$  que están dentro de  $m$  unidades de pág. Como veremos en breve, solo se considerarán los 7  $i^0$  que sigan a p puntos en  $Y$ . Calcule la distancia desde p a cada uno de estos 7 puntos, y lleve la cuenta de la distancia del par más cercano  $i0$  encontrada sobre todos los pares de puntos en  $Y$ . Si  $i0 < i^0$ , entonces la tira vertical contiene un par más cercano que el recursivo llamadas encontradas. Devuelve este par y su distancia  $i0$ . De lo contrario, devuelva el par más cercano y su distancia  $i^0$  encontrada por las llamadas recursivas.

La descripción anterior omite algunos detalles de implementación que son necesarios para lograr el tiempo de ejecución  $O(n \lg n)$ . Después de probar la corrección del algoritmo, mostraremos cómo implementar el algoritmo para lograr el límite de tiempo deseado.

#### Exactitud

La corrección de este algoritmo de par más cercano es obvia, excepto por dos aspectos.

Primero, al llegar al fondo de la recursividad cuando  $j \geq 3$ , nos aseguramos de que nunca intentaremos resolver un subproblema que consiste en un solo punto. El segundo aspecto es que solo necesitamos verificar los 7 puntos que siguen a cada punto p en la propiedad del  $i^0$ ; ahora probaremos esto arreglo  $Y$ .

Suponga que en algún nivel de la recursividad, el par de puntos más cercano es  $pL$  2  $PL$  y  $pR$  2  $PR$ . Por tanto, la distancia  $i0$  entre  $pL$  y  $pR$  es estrictamente menor que  $i^0$ .

El punto  $pL$  debe estar sobre o a la izquierda de la línea  $ly$  a menos de  $i^0$  unidades de distancia. De manera similar,  $pR$  está a la derecha o a la derecha de  $ly$  a menos de  $i^0$  unidades de distancia. Además,  $pL$  y  $pR$  están dentro de  $i^0$  unidades uno del otro verticalmente. Por lo tanto, como muestra la figura 33.11(a),  $pL$  y  $pR$  están dentro de un rectángulo  $i^0 \times 2i^0$  centrado en la línea  $l$ . (También puede haber otros puntos dentro de este rectángulo).

A continuación mostramos que como máximo 8 puntos de P pueden residir dentro de este rectángulo  $i^0 \times 2i^0$ . Considere el cuadrado  $i^0 \times i^0$  que forma la mitad izquierda de este rectángulo. Dado que todos los puntos dentro de  $PL$  están separados por al menos  $i^0$  unidades, como máximo 4 puntos pueden residir dentro de este cuadrado; La figura 33.11(b) muestra cómo. De manera similar, como máximo 4 puntos en  $PR$  pueden residir dentro del cuadrado  $i^0 \times i^0$  que forma la mitad derecha del rectángulo. Por lo tanto, como máximo 8 puntos de P pueden residir dentro del rectángulo  $i^0 \times 2i^0$ . (Tenga en cuenta que dado que los puntos en la línea  $l$  pueden estar en  $PL$  o  $PR$ , puede haber hasta 4 puntos en  $l$ . Este límite se logra si hay dos pares de puntos coincidentes, de modo que cada par consta de un punto de  $PL$  y uno desde  $PR$ , un par está en la intersección de  $l$  y la parte superior del rectángulo, y el otro par está donde  $l$  interseca la parte inferior del rectángulo).

Habiendo demostrado que como máximo 8 puntos de P pueden residir dentro del rectángulo, podemos ver fácilmente por qué necesitamos verificar solo los 7 puntos que siguen a cada punto en la matriz  $Y^{i^0}$ . Todavía suponiendo que el par más cercano es  $pL$  y  $pR$ , supongamos sin

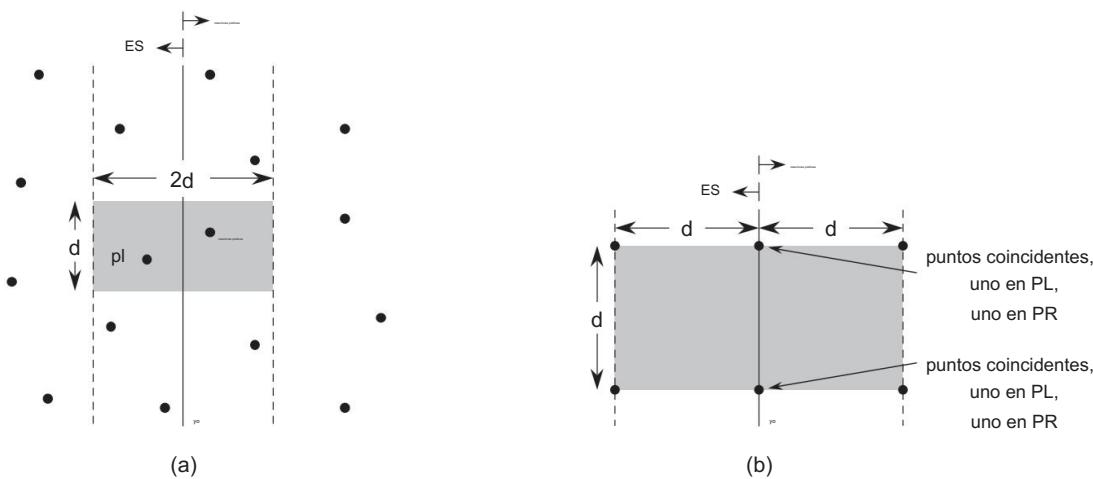


Figura 33.11 Conceptos clave en la prueba de que el algoritmo del par más cercano necesita verificar solo 7 puntos después de cada punto en el arreglo  $Y$  debe residir  $^0$ . (a) Si  $p_L$  2 PL y  $p_R$  2 PR están separados por menos de  $i$  unidades, entonces dentro de un rectángulo  $i$  2i centrado en la línea  $l$ . (b) Cómo 4 puntos que están separados por pares al menos  $i$  unidades pueden residir dentro de un cuadrado  $i$   $i$ . A la izquierda hay 4 puntos en PL ya la derecha hay 4 puntos en PR. El rectángulo  $i$   $i$  puede contener 8 puntos si los puntos que se muestran en la línea  $l$  son en realidad pares de puntos coincidentes con un punto en PL y otro en PR.

pérdida de generalidad de que  $p_L$  precede a  $p_R$  en el arreglo  $^0$ . Entonces, incluso si  $p_L$  ocurre lo antes  $Y$  como sea posible  $^0$  posible y  $p_R$  ocurre lo más tarde posible,  $p_R$  está en una de las 7 posiciones en  $Y$  siguiendo a  $p_L$ . Por lo tanto, hemos demostrado la corrección del algoritmo del par más cercano.

### Implementación y tiempo de ejecución

Como hemos señalado, nuestro objetivo es que la recursividad del tiempo de ejecución sea  $T .n/ \leq 2T .n=2/ + C On^0$ , donde  $T .n/$  es el tiempo de ejecución para un conjunto de  $n$  puntos. La principal dificultad proviene de asegurar que las matrices  $XL$ ,  $XR$ ,  $YL$  e  $YR$ , que se pasan a las llamadas recursivas, se ordenen por la coordenada adecuada y también que la matriz  $Y$  se ordene por la coordenada  $y$ . (Tenga en cuenta que si la matriz  $X$  que recibe una llamada recursiva ya está ordenada, entonces podemos dividir fácilmente el conjunto  $P$  en  $PL$  y  $PR$  en tiempo lineal).

La observación clave es que en cada llamada, deseamos formar un subconjunto ordenado de una matriz ordenada. Por ejemplo, una invocación particular recibe el subconjunto  $P$  y la matriz  $Y$  ordenada por la coordenada  $y$ . Haciendo dividido  $P$  en  $PL$  y  $PR$ , necesita formar los arreglos  $YL$  e  $YR$ , que están ordenados por la coordenada  $y$ , en tiempo lineal. Podemos ver el método como el opuesto del procedimiento MERGE de merge sort en

Sección 2.3.1: estamos dividiendo una matriz ordenada en dos matrices ordenadas. El siguiente pseudocódigo da la idea.

```

1 sean YLŒ1 : : Y:length y YRŒ1 : : Y:length nuevas matrices
2 YL:longitud D YR:longitud D 0 3 para
i D 1 a Y:longitud 4 si Y Œi 2
PL 5
    YL:longitud D YL:longitud C 1 6
    YLŒYL:longitud DY Œi
7 else YR:longitud D YR:longitud C 1 8
    YRŒYR:longitud DY Œi

```

Simplemente examinamos los puntos en el arreglo Y en orden. Si un punto Y Œi está en PL, lo agregamos al final de la matriz YL; de lo contrario, lo agregamos al final de la matriz YR.

Un pseudocódigo similar funciona para formar matrices XL, XR e Y<sup>0</sup>.

La única pregunta que queda es cómo ordenar los puntos en primer lugar. Los preclasificamos ; es decir, los ordenamos de una vez por todas antes de la primera llamada recursiva.

Pasamos estas matrices ordenadas a la primera llamada recursiva, y desde allí las reducimos a través de las llamadas recursivas según sea necesario. La clasificación previa agrega un término On lg n/ adicional al tiempo de ejecución, pero ahora cada paso de la recursividad toma un tiempo lineal exclusivo de las llamadas recursivas. Por lo tanto, si hacemos que T .n/ sea el tiempo de cada paso recursivo y T<sup>0</sup> ejecución de .n/ el tiempo de ejecución de todo el algoritmo, obtenemos T<sup>0</sup>.n/ DT .n/ C On lg n/ y

$$T .n/ \leq 2T .\frac{n}{2} + C \text{ On/ si } n > 3; \quad \text{si } n = 3$$

$$\text{Así, } T .n/ \leq 2T .\frac{n}{2} + C \text{ On lg n/}.$$

## Ejercicios

### 33.4-1

Al profesor Williams se le ocurre un esquema que permite que el algoritmo del par más cercano verifique solo 5 puntos que siguen a cada punto en el arreglo Y<sup>0</sup>. La idea siempre es colocar Y puntos en la línea l en el conjunto PL. Entonces, no puede haber pares de puntos coincidentes en la recta l con un punto en PL y otro en PR. Por lo tanto, como máximo pueden residir 6 puntos en el rectángulo I<sup>2l</sup>. ¿Cuál es la falla en el esquema del profesor?

### 33.4-2

Muestre que en realidad es suficiente verificar solo los puntos en las 5 posiciones del arreglo Y que siguen a cada punto en el arreglo Y<sup>0</sup>

## 33.4-3

Podemos definir la distancia entre dos puntos de formas distintas a la euclídea. En el plano, la distancia  $L_m$  entre los puntos  $p_1$  y  $p_2$  viene dada por la expresión  $C \sqrt{|y_1 - y_2|^m}$ . La distancia euclídea, por lo tanto, la distancia  $L_2$ , que  $\sqrt{|x_1 - x_2|^2}$  es la distancia  $L_2$ . La distancia  $L_1$ , que  $|x_1 - x_2| + |y_1 - y_2|$  Modifique el algoritmo del par más cercano para usar también se conoce como la distancia Manhattan.

## 33.4-4

Dados dos puntos  $p_1$  y  $p_2$  en el plano, la distancia  $L_1$  entre ellos está dada por  $\max(|x_1 - x_2|; |y_1 - y_2|)$ . Modifique el algoritmo del par más cercano para usar la distancia  $L_1$ .

## 33.4-5

Suponga que  $n$  de los puntos dados al algoritmo del par más cercano son ocultos.

Muestre cómo determinar los conjuntos  $PL$  y  $PR$  y cómo determinar si cada punto de  $Y$  está en  $PL$  o  $PR$  de modo que el tiempo de ejecución del algoritmo del par más cercano permanezca en  $\lg n$ .

## 33.4-6

Sugiera un cambio al algoritmo de par más cercano que evite preordenar la matriz  $Y$  pero deje el tiempo de ejecución como  $O(n \lg n)$ . (Sugerencia: combine las matrices ordenadas  $YL$  y  $YR$  para formar la matriz ordenada  $Y$ ).

## Problemas

## 33-1 Capas convexas Dado

un conjunto  $Q$  de puntos en el plano, definimos inductivamente las capas convexas de  $Q$ .

La primera capa convexa de  $Q$  consta de aquellos puntos en  $Q$  que son vértices de  $CH(Q)$ .

Para  $i > 1$ , defina  $Q_i$  para que consista en los puntos de  $Q$  con todos los puntos en capas convexas  $1; 2; \dots; i-1$  eliminado. Entonces, la  $i$ -ésima capa convexa de  $Q$  es  $CH(Q_i)$  si  $Q_i \neq \emptyset$ ; y no está definido de otra manera.

a. Proporcione un algoritmo  $O(n^2)$ -tiempo para encontrar las capas convexas de un conjunto de  $n$  puntos.

b. Demuestre que se requiere  $n \lg n$  tiempo para calcular las capas convexas de un conjunto de  $n$  puntos con cualquier modelo de cálculo que requiera  $n \lg n$  tiempo para clasificar  $n$  números reales.

## 33-2 Capas máximas Sea Q

un conjunto de  $n$  puntos en el plano. Decimos que punto  $.x; y/$  domina el punto  $.x_0 ; y_0 /$  si  $x < x_0$  e  $y < y_0$ .

Un punto en  $Q$  que no está dominado por ningún otro punto en  $Q$  se dice que es máximo. Tenga en cuenta que  $Q$  puede contener muchos puntos máximos, que se pueden organizar en capas máximas de la siguiente manera. La primera capa máxima  $L_1$  es el conjunto de puntos máximos de  $Q$ . Para  $i > 1$ , la  $i$ -ésima capa máxima  $L_i$  es el conjunto de puntos máximos en  $Q$ . Suponga que  $\{D_1 \cup L_j\}$ .

tiene  $k$  capas máximas no vacías, y sea  $y_i$  la coordenada  $y$  de  $k$ . Por ahora, suponga que no hay dos  $D_1; 2; \dots; k$  en  $Q$  tienen la misma coordenada  $x$  o  $y$ . puntos del punto más a la izquierda en  $L_i$  para  $i$

a. Demuestre que  $y_1 > y_2 > \dots > y_k$ .

Considere un punto  $.x; y/$  que está a la izquierda de cualquier punto en  $Q$  y para la cual  $y$  es distinta de la coordenada  $y$  de cualquier punto en  $Q$ . Sea  $Q_0 = D_Q \cup \{x; y\}$ . b. Sea  $j$  el índice mínimo tal

que  $y_j < y$ , a menos que  $y < y_k$ , en cuyo caso dejemos que  $j = k + 1$ . Demuestre que las capas máximas de  $Q_0$  son las siguientes:

Si  $j = k$ , entonces las capas máximas de  $Q_0$  son las mismas que las capas máximas de  $Q$ , excepto que  $L_j$  también incluye  $.x; y/$  como su nuevo punto más a la izquierda.

Si  $j > k$ , entonces las primeras  $k$  capas máximas de  $Q_0$  son las mismas que para  $Q$ , pero además,  $Q_0$  tiene una capa máxima no vacía  $.k; C_1; \dots; L_{k+1} \cup D_f; y; g.$

C. Describir un algoritmo On  $\lg n$ -time para calcular las capas máximas de un conjunto  $Q$  de  $n$  puntos. (Sugerencia: mueva una línea de barrido de derecha a izquierda).

d. ¿Surge alguna dificultad si ahora permitimos que los puntos de entrada tengan la misma  $x$ -o coordenada  $y$ ? Sugiera una manera de resolver tales problemas.

## 33-3 Cazafantasmas y fantasmas Un

grupo de  $n$  Cazafantasmas lucha contra  $n$  fantasmas. Cada Cazafantasmas lleva un paquete de protones, que dispara una corriente a un fantasma, erradicándolo. Una corriente va en línea recta y termina cuando golpea al fantasma. Los Cazafantasmas deciden la siguiente estrategia. Se emparejarán con los fantasmas, formando  $n$  parejas Cazafantasmas-fantasma, y luego, simultáneamente, cada Cazafantasmas disparará un chorro a su fantasma elegido. Como todos sabemos, es muy peligroso dejar que los arroyos se crucen, por lo que los Cazafantasmas deben elegir parejas para las que no se crucen los arroyos.

Suponga que la posición de cada Cazafantasmas y cada fantasma es un punto fijo en el plano y que no hay tres posiciones colineales.

a. Argumente que existe una línea que pasa por un Cazafantasmas y un fantasma tal que el número de Cazafantasmas en un lado de la línea es igual al número de fantasmas en el mismo lado. Describa cómo encontrar tal línea en On  $\lg n$  time.

- b. Proporcione un algoritmo  $O.n^2 \lg n$ -time para emparejar Cazafantasmas con fantasmas en tal forma en que no cruzan los arroyos.

#### 33-4 Recoger palos El profesor

Caronte tiene un conjunto de  $n$  palos, que están apilados en alguna configuración.

Cada palo está especificado por sus puntos finales, y cada punto final es un triple ordenado que da su  $x; y; z$  coordenadas. Ningún palo es vertical. Quiere recoger todos los palos, uno a la vez, sujeto a la condición de que puede recoger un palo solo si no hay otro palo encima.

- a. Proporcione un procedimiento que tome dos palos  $a$  y  $b$  e informe si  $a$  está arriba, abajo o no está relacionado con  $b$ .
- b. Describa un algoritmo eficiente que determine si es posible recoger todos los palos y, de ser así, proporcione un orden legal para recogerlos.

#### 33-5 Distribuciones de casco disperso Considere

el problema de calcular el casco convexo de un conjunto de puntos en el plano que han sido dibujados de acuerdo con alguna distribución aleatoria conocida. A veces, el número de puntos, o el tamaño, del casco convexo de  $n$  puntos extraídos de tal distribución tiene una expectativa  $O.n^{1/2}$  para alguna constante  $>0$ . Llamamos a tal distribución de casco disperso. Las distribuciones de casco disperso incluyen lo siguiente:

Puntos dibujados uniformemente a partir de un disco de radio unitario. El casco convexo tiene un tamaño esperado  $\sqrt{n}$ .

Puntos dibujados uniformemente desde el interior de un polígono convexo de  $k$  lados, para cualquier constante  $k$ . El casco convexo tiene un tamaño esperado  $\sqrt{\lg n}$ .

Puntos dibujados según una distribución normal bidimensional. El casco convexo tiene el tamaño esperado  $\sqrt{2\pi} \lg n$ .

- a. Dados dos polígonos convexos con  $n_1$  y  $n_2$  vértices respectivamente, muestre cómo calcular la envolvente convexa de todos los  $n_1n_2$  puntos en  $O.n_1n_2\lg(n_1+n_2)$  tiempo. (Los polígonos pueden superponerse).

- b. Muestre cómo calcular la envolvente convexa de un conjunto de  $n$  puntos dibujados independientemente de acuerdo con una distribución de envolvente dispersa en tiempo  $O(n\lg n)$ . (Sugerencia: busque recursivamente las envolventes convexas de los primeros  $n/2$  puntos y los segundos  $n/2$  puntos, y luego combine los resultados).

---

## Notas del capítulo

Este capítulo apenas toca la superficie de los algoritmos y técnicas de geometría computacional. Los libros sobre geometría computacional incluyen los de Preparata y Shamos [282], Edelsbrunner [99] y O'Rourke [269].

Aunque la geometría se ha estudiado desde la antigüedad, el desarrollo de algoritmos para problemas geométricos es relativamente nuevo. Preparata y Shamos señalan que la primera noción de la complejidad de un problema fue dada por E. Lemoine en 1902.

Estaba estudiando construcciones euclidianas (aquellas que usan un compás y una regla) e ideó un conjunto de cinco primitivas: colocar una pata de la brújula en un punto dado, colocar una pata de la brújula en una línea dada, dibujar un círculo, pasar el borde de la regla a través de un punto dado, y dibujar una línea. Lemoine estaba interesado en el número de elementos primitivos necesarios para efectuar una construcción dada; llamó a esta cantidad la “simplicidad” de la construcción.

El algoritmo de la Sección 33.2, que determina si algún segmento intersecta, se debe a Shamos y Hoey [313].

La versión original de la exploración de Graham está dada por Graham [150]. El algoritmo de envoltura de paquetes se debe a Jarvis [189]. Utilizando un modelo de cálculo de árbol de decisión, Yao [359] demostró un límite inferior de  $n \lg n$  en el peor de los casos para el tiempo de ejecución de cualquier algoritmo de casco convexo. Cuando se tiene en cuenta el número de vértices  $h$  de la envolvente convexa, el algoritmo de poda y búsqueda de Kirkpatrick y Seidel [206], que toma  $\lg h$  tiempo, es asintóticamente óptimo.

El algoritmo  $O(n \lg n)$ -time divide-and-conquer para encontrar el par de puntos más cercano es de Shamos y aparece en Preparata y Shamos [282]. Preparata y Shamos también muestran que el algoritmo es asintóticamente óptimo en un modelo de árbol de decisión.

---

## 34

## NP-Compleitud

Casi todos los algoritmos que hemos estudiado hasta ahora han sido algoritmos de tiempo polinomial: en entradas de tamaño  $n$ , su tiempo de ejecución en el peor de los casos es  $O.nk^l$  para alguna constante  $k$ . Quizás te preguntes si todos los problemas se pueden resolver en tiempo polinomial. La respuesta es no. Por ejemplo, hay problemas, como el famoso “problema de la detención” de Turing, que no puede resolver ningún ordenador, por mucho tiempo que le dediquemos. También hay problemas que pueden resolverse, pero no en el tiempo  $O.nk^l$  para cualquier constante  $k$ . En general, consideramos que los problemas que se pueden resolver mediante algoritmos de tiempo polinomial son tratables o fáciles, y los problemas que requieren un tiempo superpolinomial son intratables o difíciles.

El tema de este capítulo, sin embargo, es una clase interesante de problemas, llamados problemas “NP-completos”, cuyo estado se desconoce. Todavía no se ha descubierto ningún algoritmo de tiempo polinomial para un problema NP-completo, ni nadie ha podido demostrar que no pueda existir ningún algoritmo de tiempo polinomial para ninguno de ellos.

Esta llamada pregunta P  $\neq$  NP ha sido uno de los problemas de investigación abiertos más profundos y desconcertantes en la informática teórica desde que se planteó por primera vez en 1971.

Varios problemas NP-completos son particularmente tentadores porque en la superficie parecen ser similares a problemas que sabemos cómo resolver en tiempo polinomial. En cada uno de los siguientes pares de problemas, uno se puede resolver en tiempo polinomial y el otro es NP-completo, pero la diferencia entre los problemas parece ser leve:

Rutas simples más cortas frente a las más largas: en el Capítulo 24, vimos que incluso con pesos de borde negativos, podemos encontrar las rutas más cortas desde una sola fuente en un gráfico dirigido  $GD .V; E$  en  $O.VE^l$  tiempo. Sin embargo, es difícil encontrar un camino simple más largo entre dos vértices. Simplemente determinar si un gráfico contiene un camino simple con al menos un número dado de aristas es NP-completo.

Recorrido de Euler frente a ciclo hamiltoniano: un recorrido de Euler de un grafo dirigido y conectado  $GD .V; E$  es un ciclo que atraviesa cada arista de  $G$  exactamente una vez, aunque se permite visitar cada vértice más de una vez. Mediante el Problema 22-3, podemos determinar si un grafo tiene un recorrido de Euler solo en tiempo  $O(E)$  y, de hecho,

podemos encontrar los bordes del recorrido de Euler en tiempo  $O(E)$ . Un ciclo hamiltoniano de un grafo dirigido  $G = \langle V; E \rangle$  es un ciclo simple que contiene cada vértice en  $V$ .

Determinar si un gráfico dirigido tiene un ciclo hamiltoniano es NP-completo.

(Más adelante en este capítulo, probaremos que determinar si un gráfico no dirigido tiene un ciclo hamiltoniano es NP-completo).

Satisfacción de 2-CNF frente a satisfacibilidad de 3-CNF: una fórmula booleana contiene variables cuyos valores son 0 o 1; conectores booleanos como  $\wedge$  (AND),  $\vee$  (OR) y  $\neg$  (NOT); y paréntesis. Una fórmula booleana es satisfactoria si existe alguna asignación de los valores 0 y 1 a sus variables que hace que se evalúe como 1. Definiremos los términos más formalmente más adelante en este capítulo, pero informalmente, una fórmula booleana está en k-conjuntiva normal form, o k-CNF, si es el AND de cláusulas de OR de exactamente  $k$  variables o sus negaciones. Por ejemplo, la fórmula booleana  $x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_2 \wedge \neg x_3$  está en 2-CNF. (Tiene la asignación satisfactoria  $x_1 = 1; x_2 = 0; x_3 = 1$ .) Aunque podemos determinar en tiempo polinomial si una fórmula de 2-CNF es satisfactoria, veremos más adelante en este capítulo que determinar si una fórmula de 3-CNF es satisfactoria la fórmula es satisfactoria es NP-completa.

### NP-completitud y las clases P y NP

A lo largo de este capítulo, nos referiremos a tres clases de problemas: P, NP y NPC, siendo esta última clase los problemas NP-completos. Los describimos informalmente aquí, y los definiremos más formalmente más adelante.

La clase P consiste en aquellos problemas que son resolvibles en tiempo polinomial. Más específicamente, son problemas que pueden resolverse en el tiempo  $O(n^k)$  para alguna constante  $k$ , donde  $n$  es el tamaño de la entrada al problema. La mayoría de los problemas examinados en los capítulos anteriores están en P.

La clase NP consiste en aquellos problemas que son “verificables” en tiempo polinomial. ¿Qué queremos decir con que un problema sea verificable? Si de alguna manera nos dieran un “certificado” de una solución, entonces podríamos verificar que el certificado es correcto en el polinomio de tiempo en el tamaño de la entrada al problema. Por ejemplo, en el problema del ciclo hamiltoniano, dado un grafo dirigido  $G = \langle V; E \rangle$ , un certificado sería una secuencia  $h_1; h_2; \dots; h_j$  de vértices. Fácilmente podríamos comprobar en tiempo polinomial que  $i \in h_i$  para  $i = 1, 2, \dots, j$  y que  $h_1 = h_j$ .

Como otro ejemplo, para la satisfacibilidad de 3-CNF, un certificado sería una asignación de valores a las variables. Podríamos comprobar en tiempo polinomial que esta asignación satisface la fórmula booleana.

Cualquier problema en P también está en NP, ya que si un problema está en P entonces podemos resolverlo en tiempo polinomial sin siquiera proporcionar un certificado. Formalizaremos esta noción más adelante en este capítulo, pero por ahora podemos creer que P ⊆ NP. La pregunta abierta es si P = NP o no.

Informalmente, un problema está en la clase NPC, y nos referimos a él como NP completo, si está en NP y es tan "difícil" como cualquier problema en NP. Definiremos formalmente lo que significa ser tan difícil como cualquier problema en NP más adelante en este capítulo.

Mientras tanto, afirmaremos sin demostración que si cualquier problema NP-completo se puede resolver en tiempo polinomial, entonces cada problema en NP tiene un algoritmo de tiempo polinomial. La mayoría de los científicos informáticos teóricos creen que los problemas NP-completos son intratables, ya que dada la amplia gama de problemas NP-completos que se han estudiado hasta la fecha, sin que nadie haya descubierto una solución en tiempo polinomial para ninguno de ellos, sería realmente sorprendente si todos ellos podrían ser resueltos en tiempo polinomial. Sin embargo, dado el esfuerzo dedicado hasta ahora a probar que los problemas NP-completos son intratables (sin un resultado concluyente), no podemos descartar la posibilidad de que los problemas NP-completos sean, de hecho, resolubles en tiempo polinomial.

Para convertirse en un buen diseñador de algoritmos, debe comprender los rudimentos de la teoría de la integridad NP. Si puede establecer un problema como NP-completo, proporciona una buena evidencia de su intratabilidad. Como ingeniero, sería mejor dedicar su tiempo a desarrollar un algoritmo de aproximación (consulte el Capítulo 35) o resolver un caso especial manejable, en lugar de buscar un algoritmo rápido que resuelva el problema exactamente. Además, muchos problemas naturales e interesantes que en la superficie no parecen más difíciles que la clasificación, la búsqueda de gráficos o el flujo de redes son, de hecho, NP-completos. Por lo tanto, debe familiarizarse con esta notable clase de problemas.

#### Descripción general de mostrar problemas para ser NP-completos

Las técnicas que usamos para mostrar que un problema particular es NP-completo difieren fundamentalmente de las técnicas usadas en la mayor parte de este libro para diseñar y analizar algoritmos. Cuando demostramos que un problema es NP-completo, estamos afirmando lo difícil que es (o al menos lo difícil que creemos que es), en lugar de lo fácil que es. No estamos tratando de probar la existencia de un algoritmo eficiente, sino que es probable que no exista ningún algoritmo eficiente. De esta manera, las pruebas de completitud de NP guardan cierta similitud con la prueba de la Sección 8.1 de un límite inferior  $n \lg n$ -time para cualquier algoritmo de ordenación por comparación; Sin embargo, las técnicas específicas utilizadas para mostrar la completitud de NP difieren del método del árbol de decisiones utilizado en la Sección 8.1.

Nos basamos en tres conceptos clave para demostrar que un problema es NP-completo:

#### Problemas de decisión frente a problemas de optimización

Muchos problemas de interés son problemas de optimización, en los que cada solución factible (es decir, "legal") tiene un valor asociado, y deseamos encontrar una solución factible con el mejor valor. Por ejemplo, en un problema que llamamos RUTA MÁS CORTA,

nos dan un grafo no dirigido  $G$  y vértices  $u$  y deseamos encontrar un camino desde  $u$  hasta que use la menor cantidad de aristas. En otras palabras, SHORTEST-PATH es el problema de la ruta más corta de un solo par en un gráfico no dirigido y no ponderado. Sin embargo, la completitud de NP no se aplica directamente a los problemas de optimización, sino a los problemas de decisión, en los que la respuesta es simplemente "sí" o "no" (o, más formalmente, "1" o "0").

Aunque los problemas NP-completos están confinados al ámbito de los problemas de decisión, podemos aprovechar una relación conveniente entre los problemas de optimización y los problemas de decisión. Por lo general, podemos formular un problema de optimización dado como un problema de decisión relacionado al imponer un límite en el valor que se optimizará. Por ejemplo, un problema de decisión relacionado con la RUTA MÁS CORTA es la RUTA: dado un gráfico dirigido  $G$ , vértices  $u$  y un número entero  $k$ , ¿existe una ruta de  $u$  a que consta como máximo de  $k$  aristas?

La relación entre un problema de optimización y su problema de decisión relacionado funciona a nuestro favor cuando tratamos de demostrar que el problema de optimización es "difícil". Esto se debe a que el problema de decisión es, en cierto sentido, "más fácil" o, al menos, "no más difícil". Como ejemplo específico, podemos resolver la RUTA resolviendo la RUTA MÁS CORTA y luego comparando el número de aristas en la ruta más corta encontrada con el valor del parámetro  $k$  del problema de decisión. En otras palabras, si un problema de optimización es fácil, su problema de decisión relacionado también es fácil. Expresado de una manera que tiene más relevancia para la completitud de NP, si podemos proporcionar evidencia de que un problema de decisión es difícil, también brindamos evidencia de que su problema de optimización relacionado es difícil. Por lo tanto, aunque restringe la atención a los problemas de decisión, la teoría de la completitud de NP a menudo también tiene implicaciones para los problemas de optimización.

### Reducciones

La noción anterior de mostrar que un problema no es más difícil ni más fácil que otro se aplica incluso cuando ambos problemas son problemas de decisión. Aprovechamos esta idea en casi todas las pruebas de completitud de NP, de la siguiente manera. Consideraremos un problema de decisión  $A$ , que nos gustaría resolver en tiempo polinomial. Llamamos a la entrada de un problema particular una instancia de ese problema; por ejemplo, en PATH, una instancia sería un grafo particular  $G$ , vértices particulares  $u$  y de  $G$ , y un número entero  $k$ . Ahora suponga que ya sabemos cómo resolver un problema de decisión diferente  $B$  en tiempo polinomial. Finalmente, supongamos que tenemos un procedimiento que transforma cualquier instancia  $\_$  de  $A$  en alguna instancia  $\_$  de  $B$  con las siguientes características:

La transformación toma un tiempo polinomial.

Las respuestas son las mismas. Es decir, la respuesta para  $\_$  es "sí" si y solo si la respuesta para  $\_$  también es "sí".

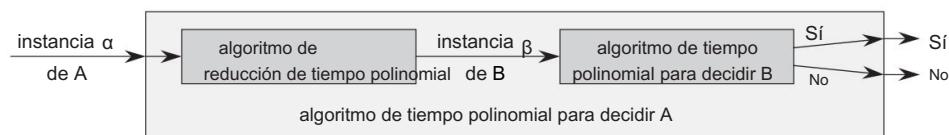


Figura 34.1 Cómo usar un algoritmo de reducción en tiempo polinómico para resolver un problema de decisión A en tiempo polinómico, dado un algoritmo de decisión en tiempo polinómico para otro problema B. En tiempo polinomial, transformamos una instancia  $\alpha$  de A en una instancia  $\beta$  de B, resolvemos B en tiempo polinomial y usamos la respuesta para  $\beta$  como la respuesta para  $\alpha$ .

A este procedimiento lo llamamos algoritmo de reducción en tiempo polinomial y, como muestra la figura 34.1, nos proporciona una forma de resolver el problema A en tiempo polinomial:

1. Dada una instancia  $\alpha$  del problema A, use un algoritmo de reducción de tiempo polinomial para transformarlo en una instancia  $\beta$  del problema B.
2. Ejecute el algoritmo de decisión de tiempo polinomial para B en la instancia  $\beta$ .
3. Usa la respuesta para  $\beta$  como la respuesta para  $\alpha$ .

Siempre que cada uno de estos pasos tome tiempo polinomial, los tres juntos también lo hacen, por lo que tenemos una manera de decidir sobre  $\alpha$  en tiempo polinomial. En otras palabras, al “reducir” la resolución del problema A a la resolución del problema B, usamos la “facilidad” de B para probar la “facilidad” de A.

Recordando que la complejidad NP se trata de mostrar qué tan difícil es un problema en lugar de qué tan fácil es, usamos reducciones de tiempo polinomial de manera opuesta para mostrar que un problema es NP-completo. Llevemos la idea un paso más allá y mostremos cómo podríamos usar reducciones de tiempo polinomial para demostrar que no puede existir ningún algoritmo de tiempo polinomial para un problema B en particular. Supongamos que tenemos un problema de decisión A para el que ya sabemos que ningún algoritmo polinomial El algoritmo de tiempo puede existir. (No nos preocupemos por ahora de cómo encontrar tal problema A). Supongamos además que tenemos una reducción de tiempo polinomial que transforma instancias de A en instancias de B. Ahora podemos usar una prueba simple por contradicción para mostrar que no existe ningún algoritmo de tiempo polinomial para B. Supongamos lo contrario; es decir, supongamos que B tiene un algoritmo de tiempo polinomial. Entonces, usando el método que se muestra en la figura 34.1, tendríamos una forma de resolver el problema A en tiempo polinomial, lo que contradice nuestra suposición de que no existe un algoritmo de tiempo polinomial para A.

Para la complejidad NP, no podemos suponer que no existe absolutamente ningún algoritmo de tiempo polinomial para el problema A. Sin embargo, la metodología de prueba es similar, ya que demostramos que el problema B es NP-completo suponiendo que el problema A también es NP-completo .

Un primer problema NP-completo

Debido a que la técnica de reducción se basa en tener un problema que ya se sabe que es NP-completo para probar un problema diferente NP-completo, necesitamos un "primer" problema NP-completo. El problema que usaremos es el problema de satisfacibilidad del circuito, en el que se nos da un circuito combinacional booleano compuesto por compuertas AND, OR y NOT, y deseamos saber si existe algún conjunto de entradas booleanas a este circuito que provoque su salida sea 1. Probaremos que este primer problema es NP-completo en la Sección 34.3.

#### Bosquejo del capítulo

Este capítulo estudia los aspectos de la completitud de NP que se relacionan más directamente con el análisis de algoritmos. En la Sección 34.1, formalizamos nuestra noción de "problema" y definimos la clase de complejidad P de los problemas de decisión solubles en tiempo polinomial. También vemos cómo estas nociones encajan en el marco de la teoría del lenguaje formal. La sección 34.2 define la clase NP de problemas de decisión cuyas soluciones son verificables en tiempo polinomial. También plantea formalmente la pregunta  $P \neq NP$ .

La sección 34.3 muestra que podemos relacionar problemas a través de "reducciones" de tiempo polinomial. Define la completitud de NP y esboza una prueba de que un problema, llamado "satisfacibilidad del circuito", es NP-completo. Habiendo encontrado un problema NP-completo, mostramos en la Sección 34.4 cómo demostrar que otros problemas son NP-completos de manera mucho más simple mediante la metodología de reducciones. Ilustramos esta metodología mostrando que dos problemas de satisfacibilidad de fórmula son NP-completos. Con reducciones adicionales, mostramos en la Sección 34.5 una variedad de otros problemas para ser NP-completos.

---

## 34.1 Tiempo polinomial

Comenzamos nuestro estudio de NP-completitud formalizando nuestra noción de problemas solubles en tiempo polinomial. En general, consideramos que estos problemas son tratables, pero por razones filosóficas, no matemáticas. Podemos ofrecer tres argumentos de apoyo mentos.

Primero, aunque razonablemente podemos considerar que un problema que requiere tiempo  $.n^{100}$  es intratable, muy pocos problemas prácticos requieren tiempo del orden de un polinomio de tan alto grado. Los problemas computables en tiempo polinomial que se encuentran en la práctica suelen requerir mucho menos tiempo. La experiencia ha demostrado que una vez que se ha descubierto el primer algoritmo de tiempo polinomial para un problema, a menudo le siguen algoritmos más eficientes. Incluso si el mejor algoritmo actual para un problema tiene un tiempo de ejecución de  $.n^{100}$ , es probable que pronto se descubra un algoritmo con un tiempo de ejecución mucho mejor.

Segundo, para muchos modelos razonables de computación, un problema que puede resolverse en tiempo polinomial en un modelo puede resolverse en tiempo polinomial en otro. Por ejemplo, la clase de problemas que se pueden resolver en tiempo polinomial con la máquina serial de acceso aleatorio utilizada en la mayor parte de este libro es la misma que la clase de problemas que se pueden resolver en tiempo polinomial en máquinas abstractas de Turing.<sup>1</sup> También es la misma que la clase de problemas resolubles en tiempo polinomial en una computadora paralela cuando el número de procesadores crece polinomialmente con el tamaño de entrada.

En tercer lugar, la clase de problemas resolubles en tiempo polinomial tiene buenos vínculos de propiedades de cierre, ya que los polinomios se cierran en la suma, la multiplicación y la composición. Por ejemplo, si la salida de un algoritmo de tiempo polinomial se alimenta a la entrada de otro, el algoritmo compuesto es polinomial. El ejercicio 34.1-5 le pide que demuestre que si un algoritmo realiza un número constante de llamadas a subrutinas de tiempo polinomial y realiza una cantidad adicional de trabajo que también requiere tiempo polinomial, entonces el tiempo de ejecución del algoritmo compuesto es polinomial.

### Problemas abstractos

Para comprender la clase de problemas resolubles en tiempo polinomial, primero debemos tener una noción formal de lo que es un "problema". Definimos un problema abstracto  $Q$  como una relación binaria en un conjunto  $I$  de instancias de problemas y un conjunto  $S$  de soluciones de problemas. Por ejemplo, una instancia de SHORTEST-PATH es un triple que consta de un gráfico y dos vértices. Una solución es una secuencia de vértices en el gráfico, tal vez con la secuencia vacía que denota que no existe un camino. El problema RUTA MÁS CORTA en sí es la relación que asocia cada instancia de un gráfico y dos vértices con una ruta más corta en el gráfico que conecta los dos vértices. Dado que las rutas más cortas no son necesariamente únicas, una instancia de problema dada puede tener más de una solución.

Esta formulación de un problema abstracto es más general de lo que necesitamos para nuestros propósitos. Como vimos anteriormente, la teoría de la completitud NP restringe la atención a los problemas de decisión: aquellos que tienen una solución sí/no. En este caso, podemos ver un problema de decisión abstracto como una función que relaciona el conjunto de instancias  $I$  con el conjunto de soluciones  $f_0: I \rightarrow S$ . Por ejemplo, un problema de decisión relacionado con la RUTA MÁS CORTA es la RUTA del problema que vimos antes. Si  $y \in D$ ,  $h(y)$  es una instancia del problema de decisión PATH, entonces  $h(y) \in D$  si y solo si el camino más corto desde  $u$  hasta  $v$  tiene como máximo  $k$  aristas, y  $h(y) \notin D$  en caso contrario. Muchos problemas abstractos no son problemas de decisión, sino problemas de optimización, que requieren que algún valor sea minimizado o maximizado. Sin embargo, como vimos anteriormente, generalmente podemos reformular un problema de optimización como un problema de decisión que no es más difícil.

---

<sup>1</sup>Véase Hopcroft y Ullman [180] o Lewis y Papadimitriou [236] para un tratamiento completo del modelo de la máquina de Turing.

### Codificaciones

Para que un programa de computadora resuelva un problema abstracto, debemos representar las instancias del problema de una manera que el programa entienda. Una codificación de un conjunto  $S$  de objetos abstractos es un mapeo  $e$  de  $S$  al conjunto de cadenas binarias.<sup>2</sup> Por ejemplo, todos estamos familiarizados con la codificación de los números naturales  $\mathbb{N}$ :  $f_0; 1; 2; 3; 4; \dots; g$  como las cadenas  $f_0; 1; 10; 11; 100; \dots; g$ . Usando esta codificación,  $e(17) = 10001$ . Si ha observado las representaciones informáticas de los caracteres del teclado, probablemente haya visto el código ASCII, donde, por ejemplo, la codificación de  $A$  es  $1000001$ . Podemos codificar un objeto compuesto como un cadena binaria combinando las representaciones de sus partes constituyentes. Polígonos, gráficos, funciones, pares ordenados, programas: todo se puede codificar como cadenas binarias.

Por lo tanto, un algoritmo de computadora que “resuelve” algún problema de decisión abstracto en realidad toma como entrada una codificación de una instancia del problema. Llamamos problema concreto a un problema cuyo conjunto de instancias es el conjunto de cadenas binarias. Decimos que un algoritmo resuelve un problema concreto en un tiempo  $O(T)$  si, cuando se le proporciona una instancia de problema  $i$  de longitud  $n$ , el algoritmo puede producir la solución en un tiempo  $O(n)$ .<sup>3</sup> Un problema concreto  $E$  es resoluble en tiempo polinomial, por lo tanto, si existe un algoritmo para resolverlo en tiempo  $O(n^k)$  para alguna constante  $k$ .

Ahora podemos definir formalmente la clase de complejidad  $P$  como el conjunto de decisiones concretas problemas de sion que son resolubles en tiempo polinomial.

Podemos usar codificaciones para mapear problemas abstractos en problemas concretos. Dado un problema de decisión abstracto  $Q$  mapeando un conjunto de instancias  $I$  a  $f_0; 1g$ , una codificación  $e$  y  $W_I : f_0; 1g \rightarrow I$  puede inducir un problema de decisión concreto relacionado, que denotaremos por  $eQ$ .

<sup>4</sup> Si la solución a una instancia de problema abstracto  $i$  es  $Q_i / 2$ , entonces la solución a la instancia del problema concreto  $e(i) / 2$  también es  $Q_i$ . Como tecnicismo, algunas cadenas binarias pueden no representar una instancia de problema abstracto significativa. Por conveniencia, supondremos que cualquier cadena de este tipo se asigna arbitrariamente a 0. Por lo tanto, el problema concreto produce las mismas soluciones que el problema abstracto en instancias de cadenas binarias que representan las codificaciones de instancias de problemas abstractos.

Nos gustaría extender la definición de resolución en tiempo polinomial de problemas concretos a problemas abstractos usando codificaciones como puente, pero nos gustaría

<sup>2</sup> El codominio de  $e$  no necesita ser cadenas binarias; cualquier conjunto de cadenas sobre un alfabeto finito que tenga al menos 2 símbolos servirá.

<sup>3</sup> Suponemos que la salida del algoritmo está separada de su entrada. Debido a que se necesita al menos un paso de tiempo para producir cada bit de la salida y el algoritmo toma  $O(n)$  pasos de tiempo, el tamaño de la salida es  $O(n)$ .

<sup>4</sup> Denotamos por  $f_0; 1g$  el conjunto de todas las cadenas compuestas por símbolos del conjunto  $f_0; 1 g$ .

gusta que la definición sea independiente de cualquier codificación en particular. Es decir, la eficiencia de resolver un problema no debería depender de cómo se codifica el problema. Desafortunadamente, depende bastante de la codificación. Por ejemplo, suponga que se va a proporcionar un número entero  $k$  como única entrada para un algoritmo, y suponga que el tiempo de ejecución del algoritmo es  $.k^k$ . Si el entero  $k$  se proporciona en unaryo, una cadena de  $k$  1, entonces el tiempo de ejecución del algoritmo es  $O(n)$  entradas de longitud  $n$ , que es tiempo polinomial. Sin embargo, si usamos la representación binaria más natural del entero  $k$ , entonces la longitud de entrada es  $\lceil \log_2 k \rceil + O(1)$ . En este caso, el tiempo de ejecución del algoritmo es  $.2^n$ , que es exponencial en el tamaño de la entrada. Por lo tanto, dependiendo de la codificación, el algoritmo se ejecuta en tiempo polinómico o superpolinomial.

La forma en que codificamos un problema abstracto es muy importante para la forma en que entendemos el tiempo polinomial. Realmente no podemos hablar de resolver un problema abstracto sin especificar primero una codificación. No obstante, en la práctica, si descartamos las codificaciones "costosas" como las unarias, la codificación real de un problema hace poca diferencia en cuanto a si el problema se puede resolver en tiempo polinomial. Por ejemplo, representar enteros en base 3 en lugar de binarios no tiene efecto sobre si un problema se puede resolver en tiempo polinomial, ya que podemos convertir un entero representado en base 3 en un entero representado en base 2 en tiempo polinomial.

Decimos que una función  $f : W \rightarrow \{0, 1\}^*$  es computable en tiempo polinomial si existe un algoritmo  $A$  en tiempo polinomial que, dada cualquier entrada  $x \in W$ , produce como salida  $f(x)$ . Para algún conjunto  $I$  de instancias de problemas, decimos que dos codificaciones  $e_1$  y  $e_2$  están relacionadas polinómicamente si existen dos funciones computables en tiempo polinomial  $f_{12}$  y  $f_{21}$  tales que para cualquier  $i \in I$  tenemos  $f_{12}.e_1.i = f_{21}.e_2.i$ .

<sup>5</sup> Es decir, un algoritmo de tiempo polinomial puede calcular la codificación  $e_2.i$  a partir de la codificación  $e_1.i$ , y viceversa. Si dos codificaciones  $e_1$  y  $e_2$  de un problema abstracto están polinomialmente relacionadas, si el problema se puede resolver en tiempo polinomial o no es independiente de la codificación que usemos, como muestra el siguiente lema.

### Lema 34.1 Sea

Q un problema de decisión abstracto sobre un conjunto de instancias  $I$ , y sean  $e_1$  y  $e_2$  codificaciones relacionadas polinómicamente sobre  $I$ . Entonces,  $e_1.Q \leq P$  si y sólo si  $e_2.Q \leq P$ .

<sup>5</sup>Técnicamente, también necesitamos las funciones  $f_{12}$  y  $f_{21}$  para "asignar no instancias a no instancias". Una no instancia de una codificación  $e$  es una cadena  $x \in \{0, 1\}^*$  tal que no hay instancia  $i$  para la cual  $e(i) = x$ . Requerimos que  $f_{12}(x) = 1$  para cada no instancia  $x$  de codificación  $e_1$ , donde  $y$  es alguna no instancia de  $e_2$ , y que  $f_{21}(y) = 1$  para cada no instancia  $y$  de  $e_2$ , donde  $x$  es alguna no instancia de  $e_1$ .

Prueba Solo necesitamos probar la dirección hacia adelante, ya que la dirección hacia atrás es simétrica. Supongamos, por tanto, que  $e1.Q/$  puede resolverse en el tiempo  $O.nk/$  para alguna constante  $k$ . Además, suponga que para cualquier instancia de problema  $i$ , la codificación  $e1.i/$  se puede calcular a partir de la codificación  $e2.i/$  en el tiempo  $O.nc/$  para alguna constante  $c$ , donde  $n \leq e2.i/j$ . Para resolver el problema  $e2.Q/$ , en la entrada  $e2.i/$ , primero calculamos  $e1.i/$  y luego ejecutamos el algoritmo para  $e1.Q/$  en  $e1.i/$ . ¿Cuánto tiempo lleva esto? La conversión de codificaciones toma tiempo  $O.nc/$ , y por lo tanto  $e1.i/j \leq O.nc/$ , ya que la salida de una computadora serial no puede ser más larga que su tiempo de ejecución. Resolviendo el en  $e1.i/$  toma tiempo  $O.e1.i/j^k$  problema /  $D.O.nc/$ , que es polinomial ya que tanto  $c$  como  $k$  son constantes. ■

Por lo tanto, si un problema abstracto tiene sus instancias codificadas en binario o en base 3, no afecta su "complejidad", es decir, si es resoluble en tiempo polinomial o no; pero si las instancias se codifican en unaryo, su complejidad puede cambiar. Para poder conversar de una manera independiente de la codificación, generalmente supondremos que las instancias del problema están codificadas de una manera razonable y concisa, a menos que digamos específicamente lo contrario. Para ser precisos, supondremos que la codificación de un entero está relacionada polinómicamente con su representación binaria, y que la codificación de un conjunto finito está relacionada polinómicamente con su codificación como una lista de sus elementos, encerrados entre llaves y separados por comas. (ASCII es uno de esos esquemas de codificación). Con una codificación "estándar" de este tipo, podemos derivar codificaciones razonables de otros objetos matemáticos, como tuplas, gráficos y fórmulas. Para indicar la codificación estándar de un objeto, encerraremos el objeto entre llaves angulares. Por lo tanto,  $hG_i$  denota la codificación estándar de un gráfico  $G$ .

Siempre que usemos implícitamente una codificación que esté polinómicamente relacionada con esta codificación estándar, podemos hablar directamente sobre problemas abstractos sin hacer referencia a ninguna codificación en particular, sabiendo que la elección de la codificación no tiene efecto sobre si el problema abstracto es resoluble en tiempo polinomial. De ahora en adelante, asumiremos generalmente que todas las instancias del problema son cadenas binarias codificadas utilizando la codificación estándar, a menos que especifiquemos explícitamente lo contrario. Típicamente también descuidaremos la distinción entre problemas abstractos y concretos. Sin embargo, debe tener cuidado con los problemas que surgen en la práctica, en los que una codificación estándar no es obvia y la codificación marca la diferencia.

#### Un marco de lenguaje formal

Al centrarnos en los problemas de decisión, podemos aprovechar la maquinaria de la teoría del lenguaje formal. Repasemos algunas definiciones de esa teoría. Un alfabeto  $\Sigma$  es un conjunto finito de símbolos. Un lenguaje  $L$  sobre  $\Sigma$  es cualquier conjunto de cadenas formado por símbolos de  $\Sigma$ . Por ejemplo, si  $\Sigma = \{0, 1\}$ , el conjunto  $L = \{f0; 1g, 11; 101; 111; 1011; 1101; 10001; \dots; g\}$  es el lenguaje de la representación binaria

taciones de números primos. Denotamos la cadena vacía con "", el idioma vacío con ; y el idioma de todas las cadenas sobre  $\emptyset$  con  $\emptyset$ . Por ejemplo, si  $\emptyset \subseteq D^*$ ; 0; 1; 00; 01; 10; 11; 000; ... : g es el conjunto de todas las cadenas binarias. Todo idioma  $L$  sobre  $\emptyset$  es un subconjunto de  $\emptyset$ .

Podemos realizar una variedad de operaciones en idiomas. Las operaciones de la teoría de conjuntos, como la unión y la intersección, se derivan directamente de las definiciones de la teoría de conjuntos. Definimos el complemento de  $L$  por  $L^c = \emptyset - L$ . La concatenación  $L_1L_2$  de dos idiomas  $L_1$  y  $L_2$  es el idioma

$L_1L_2 = \{x_1x_2 \mid x_1 \in L_1 \text{ y } x_2 \in L_2\}$

El cierre o estrella Kleene de una lengua  $L$  es la lengua

$L^* = \bigcup_{k=0}^{\infty} L^k$

el lenguaje obtenido al concatenar  $L$  consigo mismo  $k$  veces.

Desde el punto de vista de la teoría del lenguaje, el conjunto de instancias para cualquier problema de decisión  $Q$  es simplemente el conjunto  $\emptyset$ , donde  $\emptyset \subseteq D^*$ ; 1g. Dado que  $Q$  está enteramente caracterizado por aquellas instancias de problemas que producen una respuesta 1 (sí), podemos ver  $Q$  como un lenguaje  $L$  sobre  $\emptyset$   $D^*$ ; 1g, donde

$L^* = \{x \mid Q(x) = 1\}$

Por ejemplo, el problema de decisión PATH tiene el lenguaje correspondiente

$PATH = \{G \mid \exists u \in V(G) \exists v \in V(G) \exists P \subseteq E(G) \exists k \in \mathbb{N} \text{ tal que } P \text{ es un grafo no dirigido, } u \text{ y } v \text{ están en } P \text{ y } d(u, v) \leq k\}$

es un

número entero, y existe un

camino desde  $u$  hasta  $v$  en  $G$  que consiste

en como máximo  $k$  aristas:

(Cuando sea conveniente, a veces usaremos el mismo nombre, PATH en este caso, para referirnos tanto a un problema de decisión como a su lenguaje correspondiente).

El marco del lenguaje formal nos permite expresar de manera concisa la relación entre los problemas de decisión y los algoritmos que los resuelven. Decimos que un algoritmo  $A$  acepta una cadena  $x \in D^*$ ; 1g si, dada la entrada  $x$ , la salida del algoritmo  $A(x)$  es 1. El lenguaje aceptado por un algoritmo  $A$  es el conjunto de cadenas  $L_A = \{x \in D^* \mid A(x) = 1\}$ , es decir, el conjunto de cadenas que acepta el algoritmo.

Un algoritmo  $A$  rechaza una cadena  $x$  si  $A(x) = 0$ .

Incluso si el lenguaje  $L$  es aceptado por un algoritmo  $A$ , el algoritmo no necesariamente rechazará una cadena  $x \in L$  proporcionada como entrada. Por ejemplo, el algoritmo puede repetirse para siempre. Un lenguaje  $L$  es decidido por un algoritmo  $A$  si cada cadena binaria en  $L$  es aceptada por  $A$  y cada cadena binaria que no está en  $L$  es rechazada por  $A$ . Un lenguaje  $L$  es aceptado en tiempo polinomial por un algoritmo  $A$  si es aceptado por  $A$  y si además existe una constante  $k$  tal que para cualquier cuerda de longitud  $n \in \mathbb{N}$ ,  $A$  decide  $L$  en tiempo  $O(n^k)$ .

el algoritmo A acepta  $x$  en el tiempo  $O.nk^l$ . Un lenguaje L se decide en tiempo polinomial por un algoritmo A si existe una constante k tal que para cualquier cadena de longitud n el algoritmo idioma, un  $\lambda$  decide correctamente si  $x \in L$  en tiempo  $O.nk^l$ . Así,  $x \in f_0; 1g$ , para aceptar un algoritmo solo necesita producir una respuesta cuando se le proporciona una cadena en L, pero para decidir un idioma, debe aceptar o rechazar correctamente cada cadena en  $f_0; 1g$

Como ejemplo, el lenguaje PATH se puede aceptar en tiempo polinomial. Un algoritmo de aceptación de tiempo polinomial verifica que G codifica un grafo no dirigido, verifica que u y son vértices en G, usa la búsqueda primero en amplitud para calcular una ruta más corta desde u hasta G y luego compara el número de aristas en el más corto trayectoria obtenida con k. Si G codifica un gráfico no dirigido y el camino encontrado desde u hasta tiene como máximo k aristas, el algoritmo genera 1 y se detiene. De lo contrario, el ritmo del algoritmo se ejecuta para siempre. Sin embargo, este algoritmo no decide la RUTA, ya que no genera explícitamente 0 para instancias en las que la ruta más corta tiene más de k aristas.

Un algoritmo de decisión para PATH debe rechazar explícitamente cadenas binarias que no sean largas para PATH. Para un problema de decisión como PATH, dicho algoritmo de decisión es fácil de diseñar: en lugar de ejecutarse eternamente cuando no hay un camino desde u hasta con k aristas como máximo, genera 0 y se detiene. (También debe generar 0 y detenerse si la codificación de entrada es defectuosa). Para otros problemas, como el problema de detención de Turing, existe un algoritmo de aceptación, pero no existe un algoritmo de decisión.

Podemos definir informalmente una clase de complejidad como un conjunto de idiomas, cuya pertenencia está determinada por una medida de complejidad, como el tiempo de ejecución, de un algoritmo que determina si una cadena dada  $x$  pertenece al idioma L. La definición real de una clase de complejidad es algo más técnico.<sup>6</sup>

Usando este marco teórico del lenguaje, podemos proporcionar una definición alternativa de la clase de complejidad P:

PD  $f_L f_0; 1g$  W existe un algoritmo A que decide L en tiempo polinómico :

De hecho, P es también la clase de lenguajes que pueden aceptarse en tiempo polinomial.

El teorema 34.2

PD  $f_L f_W$  se acepta mediante un algoritmo de tiempo polinómico:

Prueba Debido a que la clase de lenguajes decidida por los algoritmos de tiempo polinomial es un subconjunto de la clase de lenguajes aceptados por los algoritmos de tiempo polinomial, solo necesitamos mostrar que si L es aceptado por un algoritmo de tiempo polinomial, es decidido por un algoritmo polinomial. -algoritmo de tiempo. Sea L el lenguaje aceptado por algunos

---

<sup>6</sup>Para obtener más información sobre las clases de complejidad, consulte el artículo seminal de Hartmanis y Stearns [162].

algoritmo de tiempo polinomial A. Usaremos un argumento clásico de “simulación” para construir otro algoritmo de tiempo polinomial A0 que decida L. Como A acepta L en el tiempo O.nk/ para alguna constante k, también existe una constante c tal que A acepta L en la mayoría de los pasos cnk. Para cualquier cadena de entrada x, el algoritmo A0 simula los pasos cnk de A. Después de simular los pasos cnk, el algoritmo A0 inspecciona el comportamiento de A. Si A ha aceptado x, entonces A0 acepta x generando un 1. Si A no ha aceptado x , entonces A0 rechaza x generando un 0. La sobrecarga de A0 simulando A no aumenta el tiempo de ejecución en más de un factor polinomial y, por lo tanto, A0 es un algoritmo de tiempo polinomial que decide L.

■

Tenga en cuenta que la demostración del teorema 34.2 no es constructiva. Para un lenguaje L 2 P dado, es posible que en realidad no conozcamos un límite en el tiempo de ejecución del algoritmo A que acepta L. Sin embargo, sabemos que existe dicho límite y, por lo tanto, que existe un algoritmo A0 que puede verificar el límite , aunque es posible que no podamos encontrar el algoritmo A0 fácilmente.

## Ejercicios

### 34.1-1

Defina el problema de optimización LONGEST-PATH-LENGTH como la relación que asocia cada instancia de un gráfico no dirigido y dos vértices con el número de aristas en un camino simple más largo entre los dos vértices. Defina el problema de decisión LONGEST-PATH D fhG; tu; ; ki WGD .V; E/ es un undi k 0 es un entero, y existe un grafo recto de trayectoria simple, u; 2 V de u a en G que consta de al menos CATH-LENGTH se puede , k aristasg. Demuestre que el problema de optimización LONGEST- resolver en tiempo polinomial si y solo si LONGEST-CATH 2 P.

### 34.1-2

Dé una definición formal para el problema de encontrar el ciclo simple más largo en un gráfico no dirigido. Dar un problema de decisión relacionado. Dé el lenguaje correspondiente al problema de decisión.

### 34.1-3

Proporcione una codificación formal de grafos dirigidos como cadenas binarias usando una representación de matriz de adyacencia. Haga lo mismo usando una representación de lista de adyacencia. Argumente que las dos representaciones están relacionadas polinómicamente.

### 34.1-4

¿El algoritmo de programación dinámica para el problema de la mochila 0-1 que se solicita en el ejercicio 16.2-2 es un algoritmo de tiempo polinomial? Explica tu respuesta.

## 34.1-5

Demuestre que si un algoritmo realiza como máximo un número constante de llamadas a subrutinas de tiempo polinomial y realiza una cantidad adicional de trabajo que también requiere tiempo polinomial, entonces se ejecuta en tiempo polinomial. Demuestre también que un número polinomial de llamadas a subrutinas de tiempo polinomial puede dar como resultado un algoritmo de tiempo exponencial.

## 34.1-6

Muestre que la clase P, vista como un conjunto de lenguajes, es cerrada bajo unión, intersección, concatenación, complemento y estrella de Kleene. Es decir, si  $L_1; L_2 \in P$ , luego  $L_1 \cup L_2 \in P$ ,  $L_1 L_2 \in P$ ,  $L_1^c \in P$  y  $L_1^* \in P$ .

1

## 34.2 Verificación de tiempo polinomial

Ahora analizamos los algoritmos que verifican la pertenencia a los idiomas. Por ejemplo, suponga que para un caso dado  $hG; tu; ; ki$  del problema de decisión PATH, también se nos da un camino  $p$  desde  $u$  hasta  $v$ . Podemos verificar fácilmente si  $p$  es un camino en  $G$  y si la longitud de  $p$  es como máximo  $k$ , y si es así, podemos ver  $p$  como un "certificado" de que la instancia pertenece a PATH. Para el problema de decisión PATH, este certificado no parece comprarnos mucho. Después de todo, PATH pertenece a P; de hecho, podemos resolver PATH en tiempo lineal, por lo que verificar la membresía de un certificado dado lleva tanto tiempo como resolver el problema desde cero. Ahora examinaremos un problema para el que no conocemos ningún algoritmo de decisión de tiempo polinomial y, sin embargo, dado un certificado, la verificación es fácil.

### ciclos hamiltonianos

El problema de encontrar un ciclo hamiltoniano en un grafo no dirigido se ha estudiado durante más de cien años. Formalmente, un ciclo hamiltoniano de un grafo no dirigido  $G = (V, E)$  es un ciclo simple que contiene cada vértice en  $V$ . Se dice que un gráfico que contiene un ciclo hamiltoniano es hamiltoniano; de lo contrario, es no hamiltoniano. El nombre honra a WR Hamilton, quien describió un juego matemático sobre el dodecaedro (Figura 34.2(a)) en el que un jugador clava cinco alfileres en cinco vértices consecutivos y el otro jugador debe completar el camino para formar un ciclo.

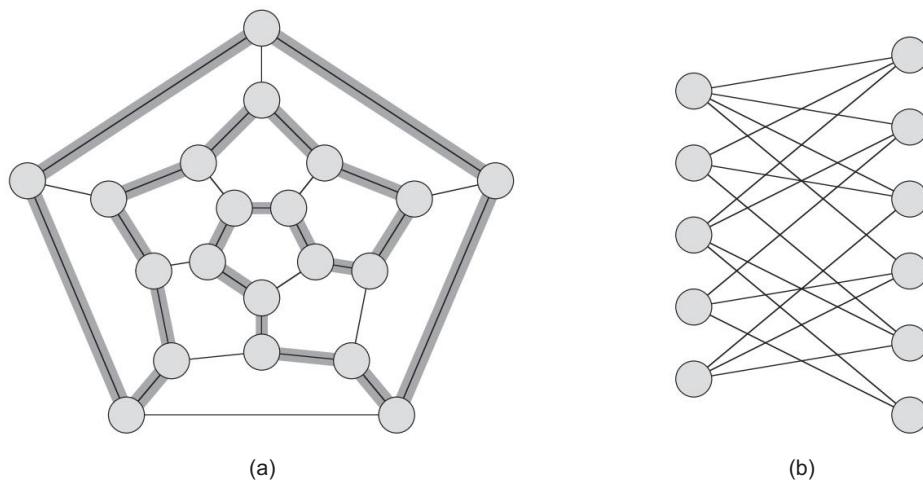


Figura 34.2 (a) Gráfico que representa los vértices, las aristas y las caras de un dodecaedro, con un ciclo hamiltoniano representado por aristas sombreadas. (b) Un gráfico bipartito con un número impar de vértices. Cualquier gráfico de este tipo es no hamiltoniano.

que contiene todos los vértices.<sup>7</sup> El dodecaedro es hamiltoniano y la figura 34.2(a) muestra un ciclo hamiltoniano. Sin embargo, no todos los gráficos son hamiltonianos. Por ejemplo, la figura 34.2(b) muestra un gráfico bipartito con un número impar de vértices.

El ejercicio 34.2-2 le pide que demuestre que todas esas gráficas son no hamiltonianas.

Podemos definir el problema del ciclo hamiltoniano, "¿Tiene un gráfico G un hamiltoniano?" como lenguaje formal:

HAM-CYCLE D fhGi WG es un gráfico hamiltoniano:

¿Cómo podría un algoritmo decidir el lenguaje HAM-CYCLE? Dada una instancia de problema hGi, un posible algoritmo de decisión enumera todas las permutaciones de los vértices de G y luego verifica cada permutación para ver si es un camino hamiltoniano. ¿Cuál es el tiempo de ejecución de este algoritmo? Si usamos la codificación “razonable” de un gráfico como su matriz de adyacencia, el número m de vértices en el gráfico es  $.pn/$ , donde n D jhGij es la longitud de la codificación de G. Hay  $m!$  permutaciones posibles

<sup>7</sup>En una carta fechada el 17 de octubre de 1856 a su amigo John T. Graves, Hamilton [157, p. 624] escribió: “He descubierto que algunos jóvenes se han divertido mucho probando un nuevo juego matemático que proporciona la Icosión, una persona clavando cinco alfileres en cinco puntos consecutivos. . . y el otro jugador entonces con el objetivo de insertar, lo que según la teoría de esta carta siempre se puede hacer, otros quince bolos, en sucesión cíclica, para cubrir todos los demás puntos, y terminar en la proximidad inmediata del bolo con el que su antagonista había empezado.”

de los vértices, y por lo tanto el tiempo de ejecución es  $m^{\overline{s}}/D \cdot p^{\overline{n}}/D \cdot 2^{\overline{pn}}$ , que no es  $O.nk/$  para ninguna constante k. Por lo tanto, este algoritmo ingenuo no se ejecuta en tiempo polinomial. De hecho, el problema del ciclo hamiltoniano es NP-completo, como demostraremos en la Sección 34.5.

### Algoritmos de verificación

Considere un problema un poco más fácil. Suponga que un amigo le dice que un grafo G dado es hamiltoniano y luego se ofrece a probarlo dándole los vértices en orden a lo largo del ciclo hamiltoniano. Sin duda, sería bastante fácil verificar la prueba: simplemente verifique que el ciclo proporcionado sea hamiltoniano verificando si es una permutación de los vértices de V y si cada una de las aristas consecutivas a lo largo del ciclo realmente existe en el gráfico. Sin duda podría implementar este algoritmo de verificación para que se ejecute en tiempo  $O.n^2/$ , donde n es la longitud de la codificación de G. Por lo tanto, una prueba de que existe un ciclo hamiltoniano en un gráfico se puede verificar en tiempo polinomial.

Definimos un algoritmo de verificación como un algoritmo A de dos argumentos, donde un argumento es una cadena de entrada ordinaria x y el otro es una cadena binaria y llamada certificado . Un algoritmo A de dos argumentos verifica una cadena de entrada x si existe un certificado y tal que  $Ax; y / D 1$ . El idioma verificado por un algoritmo de verificación A es

$\text{LD } f(x) \text{ } 2 \text{ } f_0; 1g \text{ } W \text{ existe } y \text{ } 2 \text{ } f_0; 1g \text{ tal que } Ax; y / D 1g :$

Intuitivamente, un algoritmo A verifica un lenguaje L si para cualquier cadena  $x \in L$ , existe un certificado y que A puede usar para probar que  $x \in L$ . Además, para cualquier cadena  $x \notin L$ , no debe haber ningún certificado que demuestre que  $x \in L$ . Por ejemplo, en el problema del ciclo hamiltoniano, el certificado es la lista de vértices en algún ciclo hamiltoniano. Si un gráfico es hamiltoniano, el propio ciclo hamiltoniano ofrece suficiente información para verificar este hecho. Por el contrario, si un gráfico no es hamiltoniano, no puede haber una lista de vértices que engañe al algoritmo de verificación para que crea que el gráfico es hamiltoniano, ya que el algoritmo de verificación verifica cuidadosamente el "ciclo" propuesto para estar seguro.

### La clase de complejidad NP

La clase de complejidad NP es la clase de lenguajes que pueden ser verificados por un algoritmo de tiempo polinomial.<sup>8</sup> Más precisamente, un lenguaje L pertenece a NP si y solo si existe un algoritmo de tiempo polinomial de dos entradas A y una constante c tal que

$$\text{LD } fx \ 2 \ f0; 1g \ W \text{ existe un certificado y con } jyj \leq D \cdot O(jxj) \text{ tal que } Ax; a / D \ 1g : ^c /$$

Decimos que el algoritmo A verifica el lenguaje L en tiempo polinomial.

De nuestra discusión anterior sobre el problema del ciclo hamiltoniano, ahora vemos que HAM-CYCLE  $\in$  NP. (Siempre es bueno saber que un conjunto importante no está vacío). Además, si  $L \in P$ , entonces  $L \in NP$ , ya que si hay un algoritmo de tiempo polinomial para decidir L, el algoritmo se puede convertir fácilmente en un algoritmo de verificación de dos argumentos que simplemente ignora cualquier certificado y acepta exactamente las cadenas de entrada que necesita. determina estar en L. Así,  $P \subseteq NP$ .

Se desconoce si  $P = NP$ , pero la mayoría de los investigadores creen que  $P \neq NP$  no son de la misma clase. Intuitivamente, la clase P consiste en problemas que pueden resolverse rápidamente. La clase NP consiste en problemas para los cuales se puede verificar una solución rápidamente. Es posible que haya aprendido por experiencia que a menudo es más difícil resolver un problema desde cero que verificar una solución claramente presentada, especialmente cuando se trabaja con limitaciones de tiempo. Los informáticos teóricos generalmente creen que esta analogía se extiende a las clases P y NP y, por lo tanto, que NP incluye lenguajes que no están en P.

Existe evidencia más convincente, aunque no concluyente, de que  $P \neq NP$ : la existencia de lenguajes que son "NP-completos". Estudiaremos esta clase en la Sección 34.3.

Muchas otras preguntas fundamentales más allá de la pregunta  $P \neq NP$  siguen sin resolverse. La Figura 34.3 muestra algunos escenarios posibles. A pesar del mucho trabajo de muchos investigadores, nadie sabe siquiera si la clase NP está cerrada bajo el complemento. Es decir, ¿ $L \in NP$  implica  $\overline{L} \in NP$ ? Podemos definir la clase de complejidad co-NP como el conjunto de lenguajes L tales que  $\overline{L} \in NP$ . Podemos reformular la cuestión de si NP es cerrado bajo complemento como si  $NP \subseteq co-NP$ . Como P es cerrado bajo complemento (ejercicio 34.1-6), del ejercicio 34.2-9 se deduce que  $P \subseteq NP \setminus co-NP$ . Una vez más, sin embargo, nadie sabe si  $P \subseteq NP \setminus co-NP$  o si hay algún lenguaje en  $NP \setminus co-NP$ .

<sup>8</sup>El nombre "NP" significa "tiempo polinomial no determinista". La clase NP se estudió originalmente en el contexto del no determinismo, pero este libro usa la noción algo más simple pero equivalente de verificación. Hopcroft y Ullman [180] dan una buena presentación de la completitud de NP en términos de modelos de cálculo no deterministas.

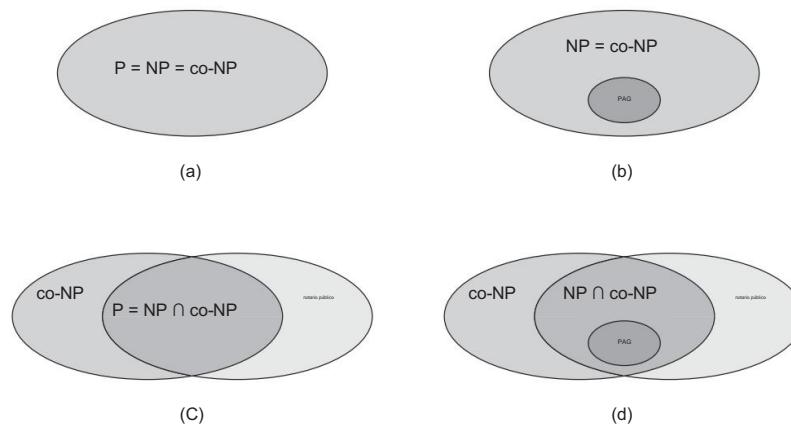


Figura 34.3 Cuatro posibilidades de relaciones entre clases de complejidad. En cada diagrama, una región que encierra a otra indica una relación de subconjunto propio. (a)  $P \subseteq NP \subseteq \text{co-NP}$ . La mayoría de los investigadores consideran esta posibilidad como la más improbable. (b) Si  $NP \subseteq \text{co-NP}$ , entonces  $NP \subseteq \text{co-NP}$ , pero no es necesario que  $P \subseteq NP$ . (c)  $P \subseteq NP \cap \text{co-NP}$ , pero  $NP \not\subseteq \text{co-NP}$ . (d)  $\text{co-NP} \not\subseteq NP \cup \text{co-NP}$ . La mayoría de los investigadores consideran esta posibilidad como la más probable.

Por lo tanto, nuestra comprensión de la relación precisa entre  $P$  y  $NP$  es completamente incompleta. Sin embargo, aunque no podamos probar que un problema en particular es intratable, si podemos probar que es  $NP$ -completo, entonces hemos obtenido información valiosa al respecto.

### Ejercicios

#### 34.2-1

Considere el lenguaje GRAFO-ISOMORFISMO. Demuestre que GRAFO-ISOMORFISMO es NP-completo describiendo un algoritmo de tiempo polinomial para verificar el lenguaje.

#### 34.2-2

Demuestre que si  $G$  es un grafo bipartito no dirigido con un número impar de vértices, entonces  $G$  no es hamiltoniano.

#### 34.2-3

Muestre que si HAM-CYCLE  $\in P$ , entonces el problema de listar los vértices de un ciclo hamiltoniano, en orden, es resoluble en tiempo polinomial.

## 34.2-4

Demuestre que la clase NP de lenguas es cerrada bajo unión, intersección, concatenación y estrella Kleene. Discuta el cierre de NP bajo complemento.

## 34.2-5

Muestre que cualquier idioma en NP puede decidirse mediante un algoritmo que se ejecuta en el tiempo  $2O.nk$  para alguna constante k.

## 34.2-6

Un camino hamiltoniano en un gráfico es un camino simple que visita cada vértice exactamente una vez. Muestre que el lenguaje HAM-PATH  $D \in \text{NP}$ ; si  $G$  tiene un camino hamiltoniano desde u hasta v en el grafo  $G$ , entonces  $G \in \text{NP}$ .

## 34.2-7

Demuestre que el problema de la trayectoria hamiltoniana del ejercicio 34.2-6 se puede resolver en tiempo polinomial en gráficas acíclicas dirigidas. Dé un algoritmo eficiente para el problema.

## 34.2-8

Sea una fórmula booleana construida a partir de las variables booleanas de entrada  $x_1; x_2; \dots; x_k$ , negaciones ( $\neg$ ), AND ( $\wedge$ ), OR ( $\vee$ ) y paréntesis. La fórmula es una tautología si se evalúa como 1 para cada asignación de 1 y 0 a las variables de entrada.

Defina TAUTOLOGÍA como el lenguaje de fórmulas booleanas que son tautologías.

Demuestre que TAUTOLOGÍA  $\in \text{NP}$ .

## 34.2-9

Demuestre que P  $\in \text{co-NP}$ .

## 34.2-10

Demuestre que si  $\text{NP} \neq \text{co-NP}$ , entonces  $\text{P} \neq \text{NP}$ .

## 34.2-11

Sea  $G$  un grafo conexo no dirigido con al menos 3 vértices, y sea  $G_3$  el grafo obtenido conectando todos los pares de vértices que están conectados por un camino en  $G$  de longitud máxima 3.

Demuestre que  $G_3$  es hamiltoniano. (Sugerencia: construya un árbol de expansión para  $G$  y use un argumento inductivo).

### 34.3 NP-completitud y reducibilidad

Quizás la razón más convincente por la que los informáticos teóricos creen que  $P \neq NP$  proviene de la existencia de la clase de problemas "NP-completos".

Esta clase tiene la intrigante propiedad de que si cualquier problema NP-completo puede resolverse en tiempo polinomial, entonces cada problema en NP tiene una solución en tiempo polinomial, es decir,  $P \in NP$ . Sin embargo, a pesar de años de estudio, nunca se ha descubierto ningún algoritmo de tiempo polinomial para ningún problema NP-completo.

El lenguaje HAM-CYCLE es un problema NP-completo. Si pudiéramos decidir HAM-CYCLE en tiempo polinomial, entonces podríamos resolver todos los problemas en NP en tiempo polinomial. De hecho, si  $NP \subseteq P$  resultara ser no vacío, podríamos decir con certeza que HAM-CYCLE  $\leq P$ .

Los lenguajes NP-completos son, en cierto sentido, los lenguajes "más difíciles" en NP. En esta sección, mostraremos cómo comparar la "dureza" relativa de los lenguajes usando una noción precisa llamada "reducibilidad de tiempo polinomial". Luego definimos formalmente los lenguajes NP-completos y terminamos esbozando una prueba de que uno de esos lenguajes, llamado CIRCUIT-SAT, es NP-completo. En las secciones 34.4 y 34.5, usaremos la noción de reducibilidad para mostrar que muchos otros problemas son NP completos.

#### Reducibilidad

Intuitivamente, un problema  $Q$  se puede reducir a otro problema  $Q_0$  si cualquier instancia de  $Q$  se puede "reformular fácilmente" como una instancia de  $Q_0$ , cuya solución proporciona una solución a la instancia de  $Q$ . Por ejemplo, el problema de resolver problemas lineales ecuaciones en una  $x$  indeterminada se reduce al problema de resolver ecuaciones cuadráticas. Dada una instancia  $ax + C = b$  ( $D = 0$ ), la transformamos en  $0x^2 + ax + C = b$  ( $D = 0$ ), cuya solución proporciona una solución a  $ax + C = b$  ( $D = 0$ ). Así, si un problema  $Q$  se reduce a otro problema  $Q_0$ , entonces  $Q$  es, en un sentido, "no más difícil de resolver" que  $Q_0$ .

Volviendo a nuestro marco de lenguaje formal para problemas de decisión, decimos que un lenguaje  $L_1$  es reducible en tiempo polinomial a un lenguaje  $L_2$ , escrito  $L_1 \leq P L_2$ , si existe una función computable en tiempo polinomial  $f : W \rightarrow f_0; 1g! f_0; 1g$  tal que para todo  $x \in f_0; 1g$ ,

$$x \in L_1 \text{ si y solo si } f(x) \in L_2 : \quad (34.1)$$

A la función  $f$  la llamamos función de reducción, y un algoritmo de tiempo polinomial  $F$  que calcula  $f$  es un algoritmo de reducción.

La figura 34.4 ilustra la idea de una reducción de tiempo polinomial de un idioma  $L_1$  a otro idioma  $L_2$ . Cada idioma es un subconjunto de  $f_0; 1g$ . La función de reducción  $f$  proporciona un mapeo de tiempo polinomial tal que si  $x \in L_1$ ,

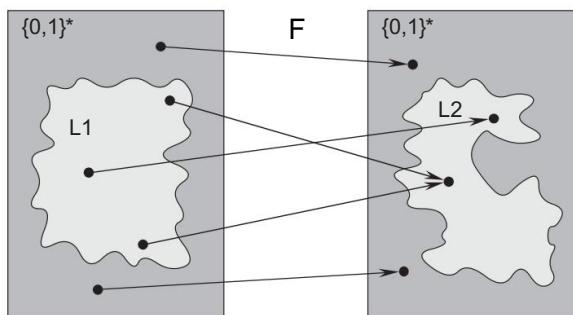


Figura 34.4 Una ilustración de una reducción de tiempo polinomial de un lenguaje  $L_1$  a un lenguaje  $L_2$  a través de una función de reducción  $f$ . Para cualquier entrada  $x \in f^{-1}(L_2)$ , la pregunta de si  $x \in L_1$  tiene la misma respuesta que la pregunta de si  $f(x) \in L_2$ .

entonces  $f(x) \in L_2$ . Además, si  $x \in L_1$ , entonces  $f(x) \in L_2$ . Así, la función de reducción mapea cualquier instancia  $x$  del problema de decisión representado por el lenguaje  $L_1$  a una instancia  $f(x)$  del problema representado por  $L_2$ . Proporcionar una respuesta a si  $f(x) \in L_2$  proporciona directamente la respuesta a si  $x \in L_1$ .

Las reducciones de tiempo polinomial nos brindan una herramienta poderosa para probar que varios lenguajes calibres pertenecen a P.

### Lema 34.3 Si

$L_1; L_2 \in P$ ;  $L_1$  y  $L_2$  son lenguajes tales que  $L_1 \leq_p L_2$ , entonces  $L_2 \in P$  implica  $L_1 \in P$ .

Demostración Sea  $A_2$  un algoritmo de tiempo polinomial que decide  $L_2$ , y sea  $F$  un algoritmo de reducción de tiempo polinomial que calcula la función de reducción  $f$ . Construiremos un algoritmo de tiempo polinomial  $A_1$  que decida  $L_1$ .

La figura 34.5 ilustra cómo construimos  $A_1$ . Para una entrada dada  $x \in f^{-1}(L_2)$ , el algoritmo  $A_1$  usa  $F$  para transformar  $x$  en  $f(x)$ , y luego usa  $A_2$  para probar si  $f(x) \in L_2$ . El algoritmo  $A_1$  toma la salida del algoritmo  $A_2$  y produce esa respuesta como su propia salida.

La corrección de  $A_1$  se sigue de la condición (34.1). El algoritmo se ejecuta en tiempo polinomial, ya que tanto  $F$  como  $A_2$  corren en tiempo polinomial (vea el Ejercicio 34.1-5). ■

### NP-completitud

Las reducciones de tiempo polinomial proporcionan un medio formal para mostrar que un problema es al menos tan difícil como otro, dentro de un factor de tiempo polinomial. Es decir, si  $L_1 \leq_p L_2$ , entonces  $L_1$  no es más que un factor polinomial más duro que  $L_2$ , que es

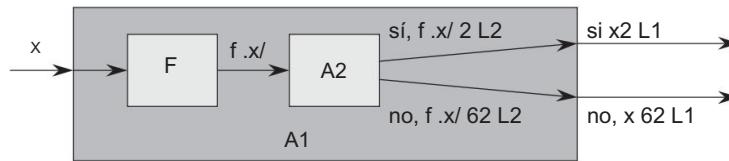


Figura 34.5 La demostración del Lema 34.3. El algoritmo F es un algoritmo de reducción que calcula la función de reducción  $f$  de  $L_1$  a  $L_2$  en tiempo polinomial, y  $A_2$  es un algoritmo de tiempo polinomial que decide  $L_2$ . El algoritmo  $A_1$  decide si  $x \in L_1$  usando  $F$  para transformar cualquier entrada  $x$  en  $f .x/$  y luego usando  $A_2$  para decidir si  $f .x/ \in L_2$ .

por qué la notación "menor que o igual a" para la reducción es mnemotécnica. Ahora podemos definir el conjunto de lenguajes NP-completos, que son los problemas más difíciles en NP.

Una lengua  $L$   $\leq_p$   $L'$  es NP-completo si

1.  $L \in NP$ , y
2.  $L \leq_p L'$  para cada  $L' \in NP$ .

Si una lengua  $L$  satisface la propiedad 2, pero no necesariamente la propiedad 1, decimos que  $L$  es NP-hard. También definimos NPC como la clase de lenguajes NP-completos.

Como muestra el siguiente teorema, la completitud de NP es el quid de la decisión si P es de hecho igual a NP.

#### Teorema 34.4 Si

cualquier problema NP-completo es resoluble en tiempo polinomial, entonces  $P = NP$ . De manera equivalente, si algún problema en NP no es resoluble en tiempo polinomial, entonces ningún problema NP-completo es resoluble en tiempo polinomial.

Prueba Suponga que  $L \in P$  y también que  $L \in NPC$ . Para cualquier  $L' \in NP$ , tenemos  $L' \leq_p L$  por la propiedad 2 de la definición de NP-completitud. Así, por el Lema 34.3, también tenemos que  $L' \in P$ , lo que prueba el primer enunciado de la teorema. ■

Para probar la segunda afirmación, tenga en cuenta que es la contrapositiva de la primera afirmación

Es por esta razón que la investigación de la pregunta  $P \neq NP$  se centra en los problemas NP-completos. La mayoría de los informáticos teóricos creen que  $P \neq NP$ , lo que conduce a las relaciones entre P, NP y NPC que se muestran en la figura 34.6.

Pero, por lo que sabemos, es posible que alguien presente un algoritmo de tiempo polinomial para un problema NP-completo, demostrando así que  $P = NP$ . Sin embargo, dado que aún no se ha descubierto ningún algoritmo de tiempo polinomial para ningún problema NP-completo,

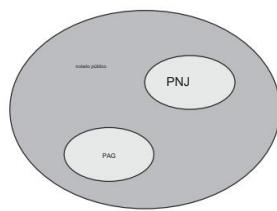


Figura 34.6 Cómo ven la mayoría de los informáticos teóricos las relaciones entre P, NP y NPC. Tanto P como NPC están completamente contenidos dentro de NP, y  $P \setminus NPC \neq \emptyset$ .

En consecuencia, una prueba de que un problema es NP-completo proporciona una excelente evidencia de que es intratable.

#### Satisfacción del circuito

Hemos definido la noción de un problema NP-completo, pero hasta este punto, en realidad no hemos probado que ningún problema sea NP-completo. Una vez que demostramos que al menos un problema es NP-completo, podemos usar la reducibilidad en tiempo polinomial como una herramienta para demostrar que otros problemas son NP-completos. Por lo tanto, ahora nos enfocamos en demostrar la existencia de un problema NP-completo: el problema de satisfacibilidad del circuito.

Desafortunadamente, la prueba formal de que el problema de satisfacibilidad del circuito es NP completo requiere detalles técnicos que van más allá del alcance de este texto. En cambio, describiremos informalmente una prueba que se basa en una comprensión básica de los circuitos nacionales combinados booleanos.

Los circuitos combinacionales booleanos se construyen a partir de elementos combinacionales booleanos que están interconectados por cables. Un elemento combinacional booleano es cualquier elemento de circuito que tiene un número constante de entradas y salidas booleanas y que realiza una función bien definida. Los valores booleanos se extraen del conjunto {0, 1}, donde 0 representa FALSO y 1 representa VERDADERO.

Los elementos combinacionales booleanos que usamos en el problema de satisfacibilidad del circuito calculan funciones booleanas simples y se conocen como puertas lógicas. La figura 34.7 muestra las tres compuertas lógicas básicas que usamos en el problema de satisfacibilidad del circuito: la compuerta NOT (o inversora), la compuerta AND y la compuerta OR. La puerta NOT toma una sola entrada binaria  $x$ , cuyo valor es 0 o 1, y produce una salida binaria  $\bar{x}$  cuyo valor es opuesto al valor de entrada. Cada una de las otras dos puertas toma dos entradas binarias  $x$  e  $y$  y produce una única salida binaria  $\bar{z}$ .

Podemos describir la operación de cada puerta y de cualquier elemento combinacional booleano mediante una tabla de verdad, que se muestra debajo de cada puerta en la figura 34.7. Una tabla de verdad da las salidas del elemento combinacional para cada configuración posible de las entradas. Para

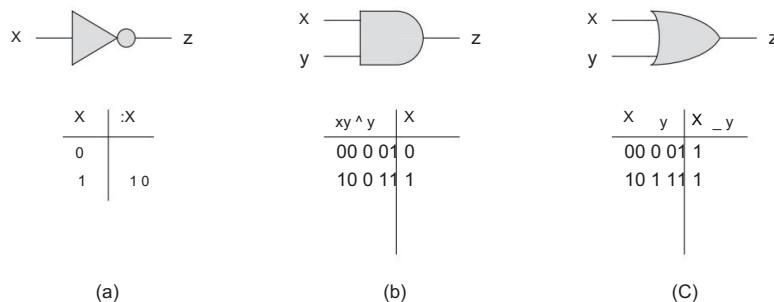


Figura 34.7 Tres puertas lógicas básicas, con entradas y salidas binarias. Debajo de cada puerta está la tabla de verdad que describe el funcionamiento de la puerta. (a) La puerta NOT. (b) La puerta AND. (c) La puerta OR.

Por ejemplo, la tabla de verdad para la puerta OR nos dice que cuando las entradas son  $x = 0$  e  $y = 1$ , el valor de salida es  $z = 1$ . Usamos los símbolos:  $\neg$  para indicar la función NOT,  $\wedge$  para indicar la función AND, y  $\vee$  para indicar la función OR. Así, por ejemplo,  $0 \vee 1 = 1$ .

Podemos generalizar las puertas AND y OR para tomar más de dos entradas. La salida de una puerta AND es 1 si todas sus entradas son 1, y su salida es 0 en caso contrario. La salida de una puerta OR es 1 si alguna de sus entradas es 1, y su salida es 0 en caso contrario.

Un circuito combinacional booleano consta de uno o más elementos combinacionales booleanos interconectados por cables. Un cable puede conectar la salida de un elemento a la entrada de otro, proporcionando así el valor de salida del primer elemento como valor de entrada del segundo. La figura 34.8 muestra dos circuitos combinacionales booleanos similares, que difieren en una sola puerta. La parte (a) de la figura también muestra los valores en los cables individuales, dada la entrada  $hx1 = 1; x2 = 1; x3 = 0$ . Aunque un solo cable no puede tener más de una salida de elemento combinacional conectada, puede alimentar varias entradas de elemento. El número de entradas de elementos alimentados por un cable se denomina fan-out del cable. Si la salida de ningún elemento está conectada a un cable, el cable es una entrada de circuito y acepta valores de entrada de una fuente externa. Si no se conecta ninguna entrada de elemento a un cable, el cable es una salida de circuito y proporciona los resultados del cálculo del circuito al mundo exterior. (Un cable interno también puede desplegarse a una salida de circuito). Con el fin de definir el problema de satisfacibilidad del circuito, limitamos el número de salidas de circuito a 1, aunque en el diseño de hardware real, un circuito combinacional booleano puede tener múltiples salidas.

Los circuitos combinacionales booleanos no contienen ciclos. En otras palabras, supongamos que creamos un grafo dirigido  $G = (V, E)$  con un vértice para cada elemento combinacional y con  $k$  aristas dirigidas para cada alambre cuyo fan-out es  $k$ ; el gráfico contiene una arista dirigida  $u \rightarrow v$  si un cable conecta la salida del elemento  $u$  con una entrada del elemento  $v$ . Entonces  $G$  debe ser acíclico.

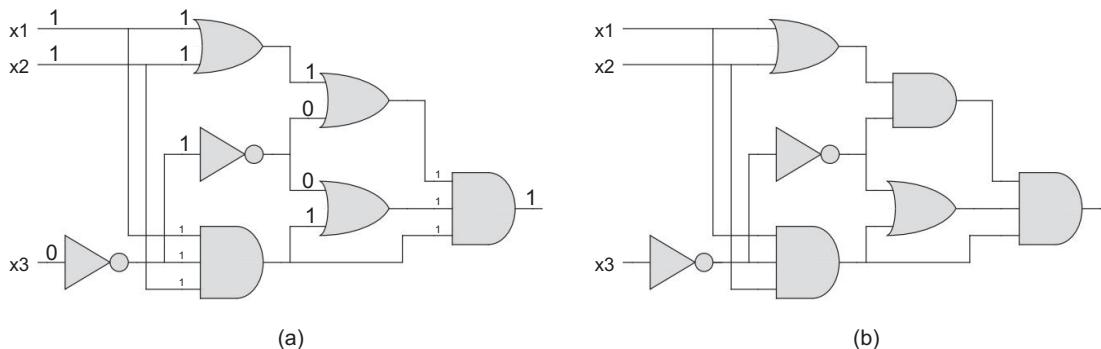


Figura 34.8 Dos instancias del problema de satisfacibilidad del circuito. (a) La asignación  $hx1 \text{ D } 1; x2 \text{ D } 1; x3 \text{ D } 0$  a las entradas de este circuito hace que la salida del circuito sea 1. Por lo tanto, el circuito es satisfactorio. (b) Ninguna asignación a las entradas de este circuito puede hacer que la salida del circuito sea 1. Por lo tanto, el circuito es insatisfactorio.

Una asignación de verdad para un circuito combinacional booleano es un conjunto de valores de entrada booleanos. Decimos que un circuito combinacional booleano de una salida es satisfactorio si tiene una asignación satisfactoria: una asignación de verdad que hace que la salida del circuito sea 1. Por ejemplo, el circuito de la figura 34.8(a) tiene la asignación satisfactoria  $hx1\ D\ 1; x2\ D\ 1; x3\ D\ 0$ , por lo que es satisfacible. Como le pide que muestre el ejercicio 34.3-1, ninguna asignación de valores a  $x_1, x_2$  y  $x_3$  hace que el circuito de la figura 34.8(b) produzca una salida 1; siempre produce 0, por lo que es insatisfactorio.

El problema de satisfacibilidad del circuito es: "Dado un circuito combinacional booleano compuesto por compuertas AND, OR y NOT, ¿es satisfacible?" Sin embargo, para plantear esta pregunta formalmente, debemos acordar una codificación estándar para los circuitos. El tamaño de un circuito combinacional booleano es el número de elementos combinacionales booleanos más el número de cables en el circuito. Podríamos idear una codificación similar a un gráfico que mapee cualquier circuito C dado en una cadena binaria  $hCi$  cuya longitud sea polinomial en el tamaño del circuito mismo. Como lenguaje formal, podemos por lo tanto definir

CIRCUIT-SAT D fhCi WC es un circuito combinacional booleano satisfactorio:

El problema de la satisfacibilidad del circuito surge en el área de optimización de hardware asistida por computadora. Si un subcircuito siempre produce 0, ese subcircuito es innecesario; el diseñador puede reemplazarlo por un subcircuito más simple que omita todas las puertas lógicas y proporcione el valor constante 0 como salida. Puede ver por qué nos gustaría tener un algoritmo de tiempo polinomial para este problema.

Dado un circuito C, podríamos intentar determinar si es satisfactorio simplemente verificando todas las asignaciones posibles a las entradas. Desafortunadamente, si el circuito tiene  $k$  entradas, entonces tendríamos que verificar hasta  $2^k$  asignaciones posibles. Cuando

el tamaño de C es polinomial en k, verificar cada uno toma  $.2k^k$  tiempo, lo cual es superpolinomio en el tamaño del circuito.<sup>9</sup> De hecho, como hemos afirmado, existe una fuerte evidencia de que no existe un algoritmo de tiempo polinomial que resuelva el problema de la satisfacibilidad del circuito porque la satisfacibilidad del circuito es NP-completo. Dividimos la prueba de este hecho en dos partes, basándonos en las dos partes de la definición de completitud NP.

#### Lema 34.5

El problema de satisfacibilidad del circuito pertenece a la clase NP.

Prueba Proporcionaremos un algoritmo A de tiempo polinomial de dos entradas que puede verificar CIRCUIT-SAT. Una de las entradas de A es (una codificación estándar de) un circuito combinado booleano C. La otra entrada es un certificado correspondiente a una asignación de valores booleanos a los cables en C. (Consulte el Ejercicio 34.3-4 para obtener un certificado más pequeño.)

Construimos el algoritmo A de la siguiente manera. Para cada puerta lógica del circuito, comprueba que el valor proporcionado por el certificado en el cable de salida se calcula correctamente en función de los valores en los cables de entrada. Entonces, si la salida de todo el circuito es 1, el algoritmo genera 1, ya que los valores asignados a las entradas de C proporcionan una asignación satisfactoria. De lo contrario, A genera 0.

Cada vez que se ingresa un circuito C satisfactorio al algoritmo A, existe un certificado cuya longitud es un polinomio del tamaño de C y que hace que A emita un 1. Cuando se ingresa un circuito insatisfactorio, ningún certificado puede engañar a A para que crea que el circuito es satisfactorio. El algoritmo A se ejecuta en tiempo polinomial: con una buena implementación, el tiempo lineal es suficiente. Así, podemos verificar CIRCUITO-SAT en tiempo polinomial, y CIRCUITO-SAT es NP. ■

La segunda parte de probar que CIRCUIT-SAT es NP-completo es demostrar que el lenguaje es NP-difícil. Es decir, debemos demostrar que todo lenguaje en NP es reducible en tiempo polinomial a CIRCUIT-SAT. La prueba real de este hecho está llena de complejidades técnicas, por lo que nos conformaremos con un esbozo de la prueba basado en cierta comprensión del funcionamiento del hardware de la computadora.

Un programa de computadora se almacena en la memoria de la computadora como una secuencia de instrucciones. Una instrucción típica codifica una operación a realizar, las direcciones de los operandos en la memoria y una dirección donde se almacenará el resultado. Una ubicación de memoria especial, llamada contador de programa, realiza un seguimiento de las instrucciones.

<sup>9</sup>Por otro lado, si el tamaño del circuito C es  $.2k^k$ , entonces un algoritmo cuyo tiempo de ejecución es  $O.2k^k$  tiene un tiempo de ejecución que es polinomial en el tamaño del circuito. Incluso si P  $\neq$  NP, esta situación no contradiría la NP-completitud del problema; la existencia de un algoritmo de tiempo polinomial para un caso especial no implica que haya un algoritmo de tiempo polinomial para todos los casos.

ción se va a ejecutar a continuación. El contador del programa aumenta automáticamente al obtener cada instrucción, lo que hace que la computadora ejecute las instrucciones secuencialmente. Sin embargo, la ejecución de una instrucción puede hacer que se escriba un valor en el contador del programa, lo que altera la ejecución secuencial normal y permite que la computadora realice un bucle y realice bifurcaciones condicionales.

En cualquier punto durante la ejecución de un programa, la memoria de la computadora mantiene todo el estado de la computación. (Tomamos la memoria para incluir el programa en sí, el contador del programa, el almacenamiento de trabajo y cualquiera de los diversos bits de estado que una computadora mantiene para su contabilidad). Llamamos configuración a cualquier estado particular de la memoria de la computadora. Podemos ver la ejecución de una instrucción como el mapeo de una configuración a otra. El hardware de la computadora que logra este mapeo se puede implementar como un circuito combinacional booleano, que denotaremos por  $M$  en la demostración del siguiente lema.

#### Lema 34.6 El

problema de satisfacibilidad del circuito es NP-difícil.

Prueba Sea  $L$  cualquier lenguaje en NP. Describiremos un algoritmo  $F$  de tiempo polinomial que calcula una función de reducción  $f$  que asigna cada cadena binaria  $x$  a un circuito  $CD f .x/$  tal que  $x \in L$  si y sólo si  $C 2 \text{CIRCUITO-SAT}$ .

Como  $L \in NP$ , debe existir un algoritmo  $A$  que verifique  $L$  en tiempo polinomial. El algoritmo  $F$  que construiremos utiliza el algoritmo  $A$  de dos entradas para calcular la función de reducción  $f$ .

Sea  $T .n/$  el peor tiempo de ejecución del algoritmo  $A$  en cadenas de entrada de longitud  $n$ , y sea  $k \geq 1$  una constante tal que  $T .n/ \leq O.nk/$  y la longitud del certificado sea  $O.nk/$ . (El tiempo de ejecución de  $A$  es en realidad un polinomio en el tamaño de entrada total, que incluye tanto una cadena de entrada como un certificado, pero dado que la longitud del certificado es polinomial en la longitud  $n$  de la cadena de entrada, el tiempo de ejecución es polinomial en  $n$ .)

La idea básica de la prueba es representar el cálculo de  $A$  como una secuencia de configuraciones. Como ilustra la figura 34.9, podemos dividir cada configuración en partes que constan del programa para  $A$ , el contador del programa y el estado de la máquina auxiliar, la entrada  $x$ , el certificado  $y$  y el almacenamiento de trabajo. El circuito combinacional  $M$ , que implementa el hardware de la computadora, asigna cada configuración  $c_i$  a la siguiente configuración  $c_{i+1}$ , comenzando desde la configuración inicial  $c_0$ . El algoritmo  $A$  escribe su salida (0 o 1) en alguna ubicación designada cuando termina de ejecutarse, y si asumimos que después  $A$  se detiene, el valor nunca cambia. Por lo tanto, si el algoritmo se ejecuta como máximo durante  $T .n/$  pasos, la salida aparece como uno de los bits en  $c_{T .n/}$ .

El algoritmo de reducción  $F$  construye un solo circuito combinacional que calcula todas las configuraciones producidas por una configuración inicial dada. La idea es

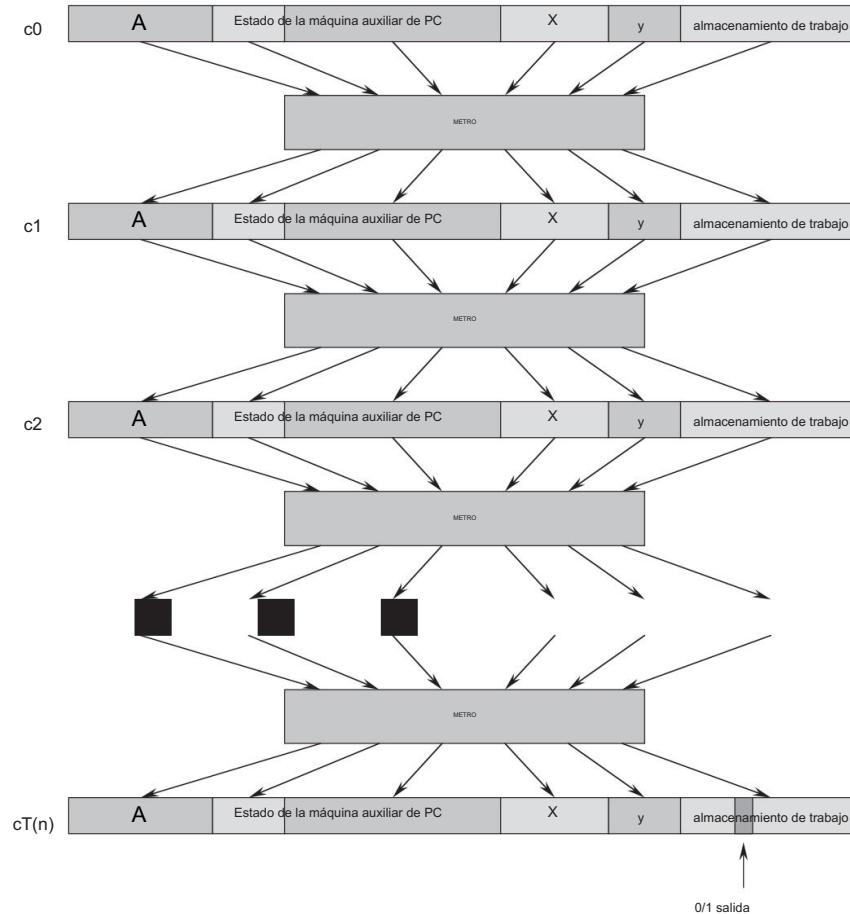


Figura 34.9 La secuencia de configuraciones producidas por un algoritmo A ejecutándose en una entrada x y un certificado y. Cada configuración representa el estado de la computadora para un paso del cálculo y, además de A, x e y, incluye el contador de programa (PC), el estado de la máquina auxiliar y el almacenamiento de trabajo. Salvo el certificado y, la configuración inicial c0 es constante. Un circuito combinacional booleano M asigna cada configuración a la siguiente configuración. La salida es un bit distinguido en el almacenamiento de trabajo.

pegue  $T \cdot n$ / copias del circuito  $M$ . La salida del  $i$ -ésimo circuito, que produce la configuración  $c_i$ , alimenta directamente la entrada del circuito  $i$ .  $C1/st$ . Por lo tanto, las configuraciones, en lugar de almacenarse en la memoria de la computadora, simplemente residen como valores en los cables que conectan las copias de  $M$ .

Recuerde lo que debe hacer el algoritmo de reducción de tiempo polinomial  $F$ . Dada una entrada  $x$ , debe computar un circuito  $CD f \cdot x/$  que es satisfacible si y sólo si existe un certificado  $y$  tal que  $Ax; y/ D 1$ . Cuando  $F$  obtiene una entrada  $x$ , primero calcula  $n D jxj$  y construye un circuito combinacional  $C0$  que consta de  $T \cdot n$ / copias de  $M$ . La entrada a  $C0$  es una configuración inicial correspondiente a un cálculo en  $Ax; y/$ , y la salida es la configuración  $cT \cdot n/$ .

El algoritmo  $F$  modifica ligeramente el circuito  $C0$  para construir el circuito  $CD f \cdot x/$ .

Primero, conecta las entradas a  $C0$  correspondientes al programa para  $A$ , el contador de programa inicial, la entrada  $x$  y el estado inicial de la memoria directamente a estos valores conocidos. Así, las únicas entradas restantes al circuito corresponden al certificado  $y$ . En segundo lugar, ignora todas las salidas de  $C0$ , excepto el bit de  $cT \cdot n/$  correspondiente a la salida de  $A$ . Este circuito  $C$ , así construido, calcula  $Cy/ D Ax; y/$  para cualquier entrada  $y$  de longitud  $O.nk/$ . El algoritmo de reducción  $F$ , cuando se le proporciona una cadena de entrada  $x$ , calcula dicho circuito  $C$  y lo genera.

Necesitamos demostrar dos propiedades. Primero, debemos demostrar que  $F$  calcula correctamente una función de reducción  $f$ . Es decir, debemos demostrar que  $C$  es satisfacible si y sólo si existe un certificado  $y$  tal que  $Ax; y/ D 1$ . Segundo, debemos mostrar que  $F$  corre en tiempo polinomial.

Para demostrar que  $F$  calcula correctamente una función de reducción, supongamos que existe un certificado  $y$  de longitud  $O.nk/$  tal que  $Ax; y/ D 1$ . Entonces, si aplicamos los bits de  $y$  a las entradas de  $C$ , la salida de  $C$  es  $Cy/ D Ax; y/ D 1$ . Por lo tanto, si existe un certificado, entonces  $C$  es satisfactorio. Para la otra dirección, suponga que  $C$  es satisfacible. Por lo tanto, existe una entrada  $y$  a  $C$  tal que  $Cy/ D 1$ , de lo cual concluimos que  $Ax; y/ D 1$ . Por lo tanto,  $F$  calcula correctamente una función de reducción.

Para completar el bosquejo de prueba, solo necesitamos mostrar que  $F$  se ejecuta en el polinomio en el tiempo en  $n D jxj$ . La primera observación que hacemos es que el número de bits necesarios para representar una configuración es polinomial en  $n$ . El programa para  $A$  en sí tiene un tamaño constante, independientemente de la longitud de su entrada  $x$ . La longitud de la entrada  $x$  es  $n$ , y la longitud del certificado  $y$  es  $O.nk/$ . Dado que el algoritmo se ejecuta como máximo en pasos  $O.nk/$ , la cantidad de almacenamiento de trabajo requerido por  $A$  también es polinomial en  $n$ .

(Suponemos que esta memoria es contigua; el ejercicio 34.3-5 le pide que extienda el argumento a la situación en la que las ubicaciones a las que accede  $A$  están dispersas en una región de memoria mucho más grande y el patrón particular de dispersión puede diferir para cada entrada  $x$ .)

El circuito combinacional  $M$  que implementa el hardware de la computadora tiene un polinomio de tamaño en la longitud de una configuración, que es  $O.nk/$ ; por lo tanto, el tamaño de  $M$  es polinomial en  $n$ . (La mayor parte de este circuito implementa la lógica de la memoria

sistema.) El circuito C consta de a lo sumo  $t D O.nk/$  copias de M, y por lo tanto tiene polinomio de tamaño en n. El algoritmo de reducción F puede construir C a partir de x en tiempo polinomial, ya que cada paso de la construcción toma tiempo polinomial. ■

El lenguaje CIRCUIT-SAT es, por lo tanto, al menos tan difícil como cualquier lenguaje en NP, y dado que pertenece a NP, es NP-completo.

### Teorema 34.7

El problema de satisfacibilidad del circuito es NP-completo.

## Prueba Inmediata de los Lemas 34.5 y 34.6 y de la definición de completitud NP. ■

### Ejercicios

#### 34.3-1

Verifique que el circuito de la figura 34.8(b) no sea satisfactorio.

#### 34.3-2

Muestre que  $L_1 \leq_p L_2$ . La relación es una relación transitiva sobre lenguajes. Es decir, demostrar que si  $L_2 \leq_p L_3$ , luego  $L_1 \leq_p L_3$ .

#### 34.3-3

Demuestre que  $L \leq_p \overline{L}$  si y solo si  $\overline{L} \leq_p L$

#### 34.3-4

Demuestre que podríamos haber usado una asignación satisfactoria como certificado en una prueba alternativa del Lema 34.5. ¿Qué certificado facilita la prueba?

#### 34.3-5

La prueba del Lema 34.6 asume que el almacenamiento de trabajo para el algoritmo A ocupa una región contigua de tamaño polinomial. ¿En qué parte de la prueba explotamos esta suposición? Argumente que esta suposición no implica ninguna pérdida de generalidad.

#### 34.3-6

Un lenguaje L es completo para una clase de lenguaje C con respecto a las reducciones de tiempo polinómico si  $L \leq_p C$  y  $L_0 \leq_p L$  para todo  $L_0 \in C$ . Demuestre que ; yf0; 1g son los únicos lenguajes en P que no están completos para P con respecto a las reducciones de tiempo polinómico.

## 34.3-7

Demuestre que, con respecto a las reducciones de tiempo polinómico (vea el ejercicio 34.3-6), L es completo para NP si y solo si L es completo para co-NP.

## 34.3-8

El algoritmo de reducción F en la prueba del Lema 34.6 construye el circuito  $CD f(x)$  basado en el conocimiento de  $x$ , A y k. El profesor Sartre observa que, pero solo la existencia de A, k y el tiempo de ejecución de  $O(nk)$ , factor constante que la cadena  $x$  se ingresa en F implícita en el es conocida por F (ya que el lenguaje L pertenece a NP), no su conocimiento real. valores. Por lo tanto, el profesor concluye que F no puede construir el circuito C y que el lenguaje CIRCUIT-SAT no es necesariamente NP-hard. Explique la falla en el razonamiento del profesor.

## 34.4 Pruebas de integridad NP

Probamos que el problema de satisfacibilidad del circuito es NP-completo mediante una demostración directa de que L CIRCUITO-SAT para cada idioma L 2 NP. En esta sección, mostraremos cómo probar que los idiomas son NP-completos sin reducir directamente cada idioma en NP al idioma dado. Ilustraremos esta metodología demostrando que varios problemas de satisfacibilidad de fórmulas son NP-completos. La Sección 34.5 proporciona muchos más ejemplos de la metodología.

El siguiente lema es la base de nuestro método para mostrar que un lenguaje es NP-completo.

### Lema 34.8

Si L es un lenguaje tal que  $L \leq_p L_0$  para algunos  $L_0$  2 NPC, entonces L es NP-hard. si, en suma, L 2 NP, entonces L 2 NPC.

Prueba Como  $L_0$  es NP-completo, para todo  $L_0$  2 NP, tenemos  $L_0 \leq_p L_0$ . Por suposición,  $L_0 \leq_p L$ , y por lo tanto por transitividad (ejercicio 34.3-2), tenemos  $L_0 \leq_p L$ , lo que muestra que L es NP-duro. Si L 2 NP, también tenemos L 2 NPC. ■

En otras palabras, al reducir un lenguaje NP-completo conocido  $L_0$  a L, implícitamente reducimos todos los lenguajes en NP a L. Por lo tanto, el Lema 34.8 nos da un método para probar que un lenguaje L es NP-completo:

1. Demuestre L 2 NP.

2. Seleccione un idioma NP-completo conocido  $L_0$ .

3. Describa un algoritmo que calcule una función  $f$  mapeando cada instancia  $x_2f_0; 1g$  de  $L_0$  a una instancia  $f .x/$  de  $L$ .
4. Demostrar que la función  $f$  satisface  $x \in L_0$  si y sólo si  $f .x/ \in L$  para todo  $x_2f_0; 1g$
5. Demuestre que el algoritmo que calcula  $f$  se ejecuta en tiempo polinomial.

(Los pasos 2 a 5 muestran que  $L$  es NP-duro). Esta metodología de reducción de un solo lenguaje NP-completo conocido es mucho más simple que el proceso más complicado de mostrar directamente cómo reducir de cada lenguaje en NP. Probar CIRCUIT-SAT 2 NPC nos ha dado un "pie en la puerta". Como sabemos que el problema de satisfacibilidad del circuito es NP-completo, ahora podemos demostrar mucho más fácilmente que otros problemas son NP-completos. Además, a medida que desarrollemos un catálogo de problemas NP-completos conocidos, tendremos más y más opciones de lenguajes a partir de los cuales reducir.

#### Satisfacción de la fórmula

Ilustramos la metodología de reducción dando una prueba de completitud NP para el problema de determinar si una fórmula booleana, no un circuito, es satisfactoria.

Este problema tiene el honor histórico de ser el primer problema que ha demostrado ser NP-completo.

Formulamos el problema de satisfacibilidad (fórmula) en términos del lenguaje SAT como sigue. Una instancia de SAT es una fórmula booleana compuesta por

1.  $n$  variables booleanas:  $x_1; x_2; \dots; x_n$ ; 2.  $m$

conectores booleanos: cualquier función booleana con una o dos entradas y una salida, como  $\wedge$  (AND),  $\vee$  (OR),  $\neg$  (NOT),  $\rightarrow$  (implicación),  $\$$  (si y sólo si); y

3. paréntesis. (Sin pérdida de generalidad, suponemos que no hay paréntesis redundantes, es decir, una fórmula contiene como máximo un par de paréntesis por conectivo booleano).

Podemos codificar fácilmente una fórmula booleana en una longitud polinomial en  $n C m$ . Al igual que en los circuitos combinacionales booleanos, una asignación de verdad para una fórmula booleana es un conjunto de valores para las variables de y una asignación satisfactoria es una asignación de verdad que hace que se evalúe como 1. Una fórmula con una asignación satisfactoria es una fórmula satisfactoria. El problema de satisfacibilidad pregunta si una fórmula booleana dada es satisfacible; en términos de lenguaje formal,

SAT D fhi W es una fórmula booleana satisfactoria:

Como ejemplo, la fórmula

$D..x1 ! x2/ \dots :x1 \$ x3/ \dots x4// ^ :x2$

tiene la asignación satisfactoria  $hx1 D 0; x2 D 0; x3 D 1; x4 D 1$ , ya que

$$\begin{aligned} &D..0! 0/ \dots :0 \$ 1/ \dots 1// ^ :0 D .1 \dots 1 // ^ 1 D \\ &.1 \dots 0 / ^ 1 D 1 \end{aligned} \tag{34.2}$$

:

y por lo tanto esta fórmula pertenece al SAT.

El algoritmo ingenuo para determinar si una fórmula booleana arbitraria es satisfactoria no se ejecuta en tiempo polinomial. Una fórmula con  $n$  variables tiene  $2^n$  asignaciones posibles. Si la longitud de  $hi$  es un polinomio en  $n$ , entonces verificar cada asignación requiere  $.2^n$  tiempo, que es un superpolinomio en la longitud de  $hi$ . Como muestra el siguiente teorema, es poco probable que exista un algoritmo de tiempo polinomial.

#### Teorema 34.9

La satisfacción de las fórmulas booleanas es NP-completa.

Prueba Empezamos argumentando que SAT  $\in$  NP. Entonces probamos que SAT es NP-duro demostrando que CIRCUITO-SAT  $\leq$  SAT; por el Lema 34.8, esto probará el teorema.

Para mostrar que SAT pertenece a NP, mostramos que un certificado que consiste en una asignación satisfactoria para una fórmula de entrada se puede verificar en tiempo polinomial. El algoritmo de verificación simplemente reemplaza cada variable en la fórmula con su valor correspondiente y luego evalúa la expresión, tal como lo hicimos en la ecuación (34.2) anterior. Esta tarea es fácil de hacer en tiempo polinomial. Si la expresión se evalúa como 1, entonces el algoritmo ha verificado que la fórmula es satisfactoria. Por lo tanto, se cumple la primera condición del Lema 34.8 para la completitud NP.

Para demostrar que SAT es NP-duro, demostramos que CIRCUIT-SAT  $\leq$  SAT. En otras palabras, necesitamos mostrar cómo reducir cualquier instancia de satisfacibilidad de circuito a una instancia de satisfacibilidad de fórmula en tiempo polinomial. Podemos usar la inducción para expresar cualquier circuito combinacional booleano como una fórmula booleana. Simplemente observamos la puerta que produce la salida del circuito y expresamos inductivamente cada una de las entradas de la puerta como fórmulas. Luego obtenemos la fórmula para el circuito escribiendo una expresión que aplica la función de la puerta a las fórmulas de sus entradas.

Desafortunadamente, este método sencillo no equivale a una reducción de tiempo polinomial. Como se le pide que muestre en el ejercicio 34.4-1, las subfórmulas compartidas, que surgen de puertas cuyos cables de salida tienen un abanico de 2 o más, pueden hacer que el tamaño de la fórmula generada crezca exponencialmente. Por lo tanto, el algoritmo de reducción debe ser algo más inteligente.

La figura 34.10 ilustra cómo superamos este problema, utilizando como ejemplo el circuito de la figura 34.8(a). Para cada cable  $x_i$  en el circuito  $C$ , la fórmula

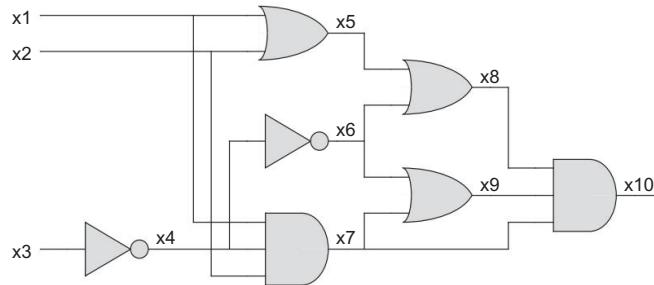


Figura 34.10 Reducción de la satisfacibilidad del circuito a la satisfacibilidad de la fórmula. La fórmula producida por el algoritmo de reducción tiene una variable para cada cable del circuito.

tiene una variable  $x_i$ . Ahora podemos expresar cómo funciona cada puerta como una pequeña fórmula que involucra las variables de sus cables incidentes. Por ejemplo, la operación de la compuerta AND de salida es  $x_{10} \equiv x_3 \wedge x_5 \wedge x_8 \wedge x_9$ . Llamamos a cada una de estas pequeñas fórmulas una cláusula.

La fórmula producida por el algoritmo de reducción es el AND de la variable de salida del circuito con la conjunción de cláusulas que describen el funcionamiento de cada puerta. Para el circuito de la figura, la fórmula es

$$\begin{aligned}
 & D \ x_{10} \wedge x_4 \equiv x_3 \wedge x_5 \\
 & \quad \wedge x_1 \wedge x_2 \wedge x_6 \equiv x_4 \\
 & \quad \wedge x_7 \equiv x_1 \wedge x_2 \wedge \\
 & \quad x_4 \wedge x_8 \equiv x_5 \wedge x_6 \wedge x_9 \\
 & \quad \wedge x_6 \wedge x_7 \wedge x_{10} \equiv x_7 \\
 & \quad \wedge x_8 \wedge x_9 \wedge :
 \end{aligned}$$

Dado un circuito C, es sencillo producir tal fórmula en tiempo polinomial.

¿Por qué el circuito C es satisfactorio exactamente cuando la fórmula es satisfactoria? Si C tiene una asignación satisfactoria, entonces cada cable del circuito tiene un valor bien definido y la salida del circuito es 1. Por lo tanto, cuando asignamos valores de cable a variables en cada cláusula de se evalúa como 1 y, por lo tanto, la conjunción de todos se evalúa como 1. Por el contrario, si alguna asignación hace que se evalúe como 1, el circuito C se puede satisfacer mediante un argumento análogo. Así, hemos demostrado que CIRCUIT-SAT

$\leq P$  SAT, que completa la prueba. ■

### Satisfacción 3-CNF

Podemos probar muchos problemas NP-completos al reducir la satisfacibilidad de la fórmula. Sin embargo, el algoritmo de reducción debe manejar cualquier fórmula de entrada, y este requisito puede dar lugar a una gran cantidad de casos que debemos considerar. A menudo preferimos reducir de un lenguaje restringido de fórmulas booleanas, por lo que necesitamos considerar menos casos. Por supuesto, no debemos restringir tanto el lenguaje como para que sea resoluble en tiempo polinomial. Un lenguaje conveniente es la satisfacción 3-CNF, o 3-CNF-SAT.

Definimos la satisfacibilidad de 3-CNF utilizando los siguientes términos. Un literal en una fórmula booleana es una ocurrencia de una variable o su negación. Una fórmula booleana está en forma normal conjuntiva, o CNF, si se expresa como un AND de cláusulas, cada una de las cuales es el OR de uno o más literales. Una fórmula booleana está en forma normal de 3 conjunciones, o 3-CNF, si cada cláusula tiene exactamente tres literales distintos.

Por ejemplo, la fórmula booleana

$.x1\_ :x1\_ :x2/ \wedge .x3\_ x2\_ x4/ \wedge .:x1\_ :x3\_ :x4/$

está en 3-CNF. La primera de sus tres cláusulas es  $.x1\_ :x1\_ :x2/$ , que contiene los tres literales  $x1, :x1$  y  $:x2$ .

En 3-CNF-SAT, se nos pregunta si una fórmula booleana dada en 3-CNF es satisfactoria. El siguiente teorema muestra que es poco probable que exista un algoritmo de tiempo polinomial que pueda determinar la satisfacibilidad de fórmulas booleanas, incluso cuando se expresan en esta forma normal simple.

#### El teorema 34.10

La satisfacibilidad de fórmulas booleanas en forma normal 3-conjuntiva es NP-completa.

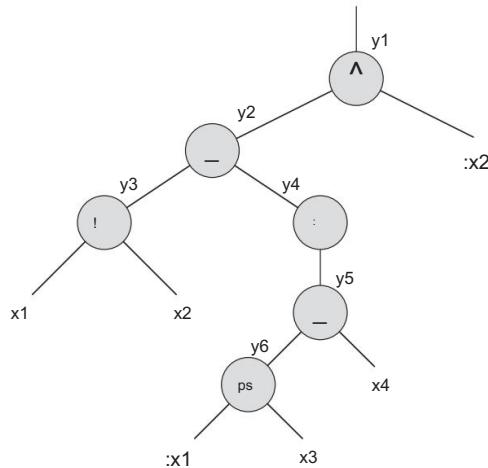
Demostración El argumento que usamos en la demostración del teorema 34.9 para mostrar que SAT  $\leq_{NP}$  3-CNF-SAT se aplica igualmente bien aquí para demostrar que 3-CNF-SAT  $\leq_{NP}$  SAT. Por el Lema 34.8, por lo tanto, solo necesitamos mostrar que SAT  $\leq_{NP}$  3-CNF-SAT.

Dividimos el algoritmo de reducción en tres pasos básicos. Cada paso progresivamente transforma la fórmula de entrada más cerca de la forma normal de 3 conjunciones deseada.

El primer paso es similar al utilizado para demostrar CIRCUITO-SAT SAT en el Teorema 34.9. Primero, construimos un árbol de "análisis" binario para la fórmula de entrada con literales , como hojas y conectores como nodos internos. La figura 34.11 muestra un árbol de análisis sintáctico para la fórmula

$$D..x1 ! x2/ \_ ...:x1 \$ x3/ \_ x4// \wedge :x2 : \quad (34.3)$$

Si la fórmula de entrada contiene una cláusula como el OR de varios literales, usamos la asociatividad para poner la expresión entre paréntesis por completo, de modo que cada nodo interno en el árbol resultante tenga 1 o 2 hijos. Ahora podemos pensar en el árbol de análisis binario como un circuito para calcular la función.

Figura 34.11 El árbol correspondiente a la fórmula  $D \ldots x1!x2/\_:\ldots:x1\$x3/_x4//^:x2:$ 

Imitando la reducción en la demostración del teorema 34.9, introducimos una variable  $y_i$  para la salida de cada nodo interno. Luego, reescribimos la fórmula original como el AND de la variable raíz y una conjunción de cláusulas que describen la operación de cada nodo. Para la fórmula (34.3), la expresión resultante es

$$\begin{aligned}
 & D \ y1 \wedge .y1 \$ .y2 \wedge :x2// \wedge .y2 \$ .y3 \\
 & \quad _ y4// \wedge .y3 \$ .x1 ! x2// \\
 & \quad ^ .y4 \$ :y5/ \wedge .y5 \$ .y6 _ \\
 & \quad x4// \wedge .y6 \$ .:x1 \\
 & \quad \$ x3// :
 \end{aligned}$$

Obsérvese que la fórmula así obtenida es una conjunción de cláusulas cada una de las cuales tiene como máximo 3 literales. El único requisito que podríamos no cumplir es que cada cláusula debe ser un OR de 3 literales.

El segundo paso de la reducción convierte cada cláusula  $\bigvee_i$  en forma conjuntiva normal. Construimos una tabla de verdad  $\bigvee_i$  evaluando todas las posibles asignaciones a sus variables. Cada fila de la tabla de verdad consta de una posible asignación de las variables de la cláusula, junto con el valor de la cláusula bajo esa asignación. Usando las entradas de la tabla de verdad que se evalúan como 0, construimos una fórmula en forma normal disyuntiva (o DNF), un OR de AND, que es equivalente a  $\bigvee_i$ . Luego negamos esta fórmula y la convertimos en una fórmula CNF usando la fórmula de DeMorgan.

$y_1 \ y_2 \ x_2 \ .y_1 \ \$ \ .y_2 \ ^ \ :x_2 // \ 111 \ 0 \ 110 \ 1$
101 0
100 0
011 1
010 0
001 1
000 1

Figura 34.12 La tabla de verdad para la cláusula  $.y_1 \ \$ \ .y_2 \ ^ \ :x_2 //$ .

leyes de la lógica proposicional,

$$\begin{aligned} &.:a \ ^ \ b / D :a \ _ \ :b \ .:a \ _ \ b / D :a \ : \\ &\ ^ \ :b \end{aligned}$$

para complementar todos los literales, cambie los OR por AND y cambie los AND por OR.

En nuestro ejemplo, convertimos la cláusula de la  $\stackrel{0}{\text{D}} \stackrel{1}{\text{:y}_1 \ \$ \ .y_2 \ ^ \ :x_2 //}$  en CNF 1 aparece en siguiente manera. La tabla de verdad  $\stackrel{0}{\text{figura 34.12}}$  la figura 34.12. La fórmula DNF para el equivalente a  $\stackrel{1}{\text{0}}$  es

$$.:y_1 \ ^ \ y_2 \ ^ \ x_2 / \ _ \ .y_1 \ ^ \ :y_2 \ ^ \ x_2 / \ _ \ .y_1 \ ^ \ :y_2 \ ^ \ :x_2 / \ _ \ .:y_1 \ ^ \ y_2 \ ^ \ :x_2 / :$$

Negando y aplicando las leyes de DeMorgan, obtenemos la fórmula CNF

$$\begin{aligned} &\stackrel{00}{1} \stackrel{0}{\text{D}} \ .:y_1 \ _ \ :y_2 \ _ \ :x_2 / \ ^ \ .:y_1 \ _ \ y_2 \ _ \ :x_2 / \ ^ \ .:y_1 \ _ \ y_2 \ _ \ x_2 / \\ &\ ^ \ .:y_1 \ _ \ :y_2 \ _ \ x_2 / ; \end{aligned}$$

que es equivalente a la cláusula original

En este punto, hemos convertido cada cláusula en una fórmula CNF  $\stackrel{0}{\text{y}}$ , por lo tanto, es equivalente  $\stackrel{00}{y_1}$  a la fórmula CNF que consiste en la conjunción de  $\stackrel{0}{i}$ . Además, cada cláusula de tiene como máximo 3 literales.

El tercer y último paso de la reducción transforma aún más la fórmula para que cada cláusula tiene exactamente 3 literales distintos. Construimos la fórmula 3-CNF final a partir de las cláusulas de la fórmula 00 de CNF . La fórmula también utiliza dos variables auxiliares que llamaremos p y q. Para cada cláusula Ci de 00, incluimos las siguientes cláusulas en 000:

Si Ci tiene 3 literales distintos, simplemente incluya Ci como una cláusula de 000.

Si Ci tiene 2 literales distintos, es decir, si Ci D .I1 \_ I2/, donde I1 y I2 son literales, entonces incluya .I1 \_ I2 \_ p/ ^ .I1 \_ I2 \_ :p/ como cláusulas de 000. Los literales p y :p simplemente cumplen el requisito sintáctico de que cada cláusula de tiene

exactamente 3 literales distintos. Ya sea  $p \oplus 0$  o  $p \oplus 1$ , una de las cláusulas es equivalente a  $I_1 \wedge I_2$ , y la otra se evalúa como 1, que es la identidad de AND.

Si  $C_i$  tiene solo 1 literal  $I$  distinto, entonces incluya  $.I \wedge p \wedge q / \wedge .I \wedge p \wedge \neg q / \wedge .I \wedge \neg p \wedge q / \wedge .I \wedge \neg p \wedge \neg q /$  como cláusulas de 000 Independientemente de los valores de  $p$  y  $q$ , una de las cuatro cláusulas es equivalente a  $I$ , y las otras 3 evalúan a 1.

Podemos ver que la fórmula 3-CNF es satisfactoria si y sólo si es satisfactoria mediante la inspección de cada uno de los tres pasos. Al igual que la reducción de CIRCUITO-SAT a SAT, la construcción de desde en el primer paso conserva la satisfacibilidad. El segundo paso produce una fórmula CNF. El tercer paso produce una fórmula que es algebraicamente equivalente a 00, ya que cualquier asignación a las variables  $p$  y  $q$  produce una fórmula que es algebraicamente equivalente a 00.

También debemos demostrar que la reducción se puede calcular en tiempo polinomial. Construir a partir de introduce como máximo 1 variable y 1 cláusula por conectivo en Construir a partir de puede introducir como máximo 8 cláusulas en para cada cláusula desde ya que cada cláusula de tiene como máximo 3 variables, y la tabla de verdad para cada cláusula tiene como máximo 23 filas La construcción de from introduce para como máximo 4 cláusulas cada cláusula de 00. Por lo tanto, el tamaño de la resultante es en fórmula polinomial en la longitud de la fórmula original. Cada una de las construcciones se puede realizar fácilmente en tiempo polinomial. ■

## Ejercicios

### 34.4-1

Considere la reducción directa (tiempo no polinomial) en la prueba del teorema 34.9. Describa un circuito de tamaño  $n$  que, cuando se convierta en una fórmula por este método, produzca una fórmula cuyo tamaño sea exponencial en  $n$ .

### 34.4-2

Muestre la fórmula 3-CNF que resulta cuando usamos el método del Teorema 34.10 en la fórmula (34.3).

### 34.4-3

El profesor Jagger propone demostrar que SAT 3-CNF-SAT usando solo la técnica de la tabla de verdad en la demostración del teorema 34.10, y no los otros pasos. Es decir, el profesor propone tomar la fórmula booleana de una tabla de verdad para sus variables, derivar de la tabla de verdad una fórmula en 3-DNF que sea equivalente a  $\perp$ , y luego negar y aplicar las leyes de DeMorgan para producir una fórmula 3-CNF equivalente a  $\perp$ . Demuestre que esta estrategia no produce una reducción de tiempo polinomial.

## 34.4-4

Muestre que el problema de determinar si una fórmula booleana es una tautología es completo para co-NP. (Sugerencia: vea el ejercicio 34.3-7.)

## 34.4-5

Demuestre que el problema de determinar la satisfacibilidad de fórmulas booleanas en forma normal disyuntiva es resoluble en tiempo polinomial.

## 34.4-6

Suponga que alguien le da un algoritmo de tiempo polinomial para decidir la satisfacibilidad de la fórmula. Describa cómo usar este algoritmo para encontrar asignaciones satisfactorias en tiempo polinomial.

## 34.4-7

Sea 2-CNF-SAT el conjunto de fórmulas booleanas satisfactorias en CNF con exactamente 2 literales por cláusula. Demuestre que 2-CNF-SAT  $\leq P$ . Haga que su algoritmo sea lo más eficiente posible. (Sugerencia: observe que  $x \wedge \neg y$  es equivalente a  $\neg x \vee y$ . Reduzca 2-CNF-SAT a un problema que se pueda resolver de manera eficiente en un gráfico dirigido).

## 34.5 Problemas NP-completos

Los problemas NP-completos surgen en diversos dominios: lógica booleana, gráficos, aritmética, diseño de redes, conjuntos y particiones, almacenamiento y recuperación, secuenciación y programación, programación matemática, álgebra y teoría de números, juegos y rompecabezas, autómatas y teoría del lenguaje, programación, optimización, biología, química, física y más. En esta sección, utilizaremos la metodología de reducción para proporcionar pruebas de completitud de NP para una variedad de problemas extraídos de la teoría de grafos y la partición de conjuntos.

La figura 34.13 describe la estructura de las pruebas de completitud NP en esta sección y en la sección 34.4. Demostramos que cada idioma en la figura es NP-completo por reducción del idioma que lo señala. En la raíz está CIRCUITO-SAT, que demostramos NP-completo en el Teorema 34.7.

### 34.5.1 El problema de la camarilla

Una camarilla en un grafo no dirigido  $G = (V, E)$  es un subconjunto  $V'$  de vértices, cada par de los cuales está conectado por una arista en  $E$ . En otras palabras, una camarilla es un subgrafo completo de  $G$ . El tamaño de una camarilla es el número de vértices que contiene. El problema de la camarilla es el problema de optimización de encontrar una camarilla de tamaño máximo en

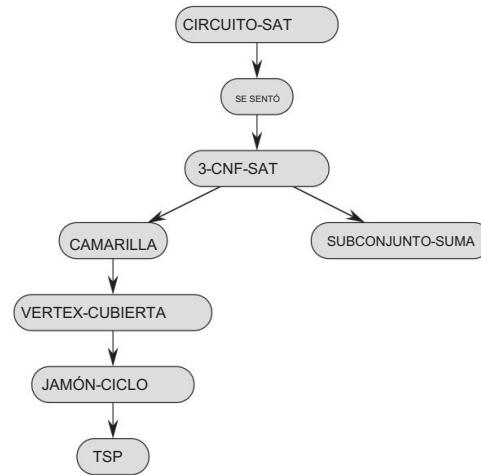


Figura 34.13 La estructura de las pruebas de completitud NP en las Secciones 34.4 y 34.5. Todas las pruebas siguen finalmente por reducción de la NP-completitud de CIRCUIT-SAT.

un gráfico. Como problema de decisión, preguntamos simplemente si existe una camarilla de un tamaño  $k$  dado en el gráfico. La definición formal es

camarilla  $D \models hG; k \models WG$  es un gráfico que contiene una camarilla de tamaño  $k$  :

Un algoritmo ingenuo para determinar si un gráfico  $G \models V; E$  con  $jV$  vértices tiene una camarilla de tamaño  $k$  es hacer una lista de todos los  $k$ -subconjuntos de  $V$  y verificar cada uno para ver si forma una camarilla. El tiempo de ejecución de este algoritmo es  $.k^2 j$ , que es polinomial si  $k$  es una constante. Sin embargo, en general,  $k$  podría estar cerca de  $jV$  ( $j = 2$ , en cuyo caso el algoritmo se ejecuta en tiempo superpolinomial). De hecho, es poco probable que exista un algoritmo eficiente para el problema de la camarilla.

Teorema 34.11 El

problema de la camarilla es NP-completo.

Prueba Para demostrar que CLIQUE  $\in$  NP, para un grafo dado  $G \models V; E$ , usamos el conjunto  $V$

$V^0$  de vértices en la camarilla como un certificado para  $G$ . Podemos comprobar si  $V^0$  es una camarilla en tiempo polinomial comprobando si, para cada par  $u; v \in V^0$ , el borde  $uv$  pertenece a  $E$ .

Probamos a continuación que  $3\text{-CNF-SAT} \leq P \text{ CLIQUE}$ , que muestra que el problema de la camarilla es NP-difícil. Es posible que se sorprenda de que podamos probar tal resultado, ya que, en la superficie, las fórmulas lógicas parecen tener poco que ver con los gráficos.

El algoritmo de reducción comienza con una instancia de 3-CNF-SAT. Sea  $D = C_1 \wedge C_2 \wedge \dots \wedge C_k$  una fórmula booleana en 3-CNF con  $k$  cláusulas, para  $r \in D$

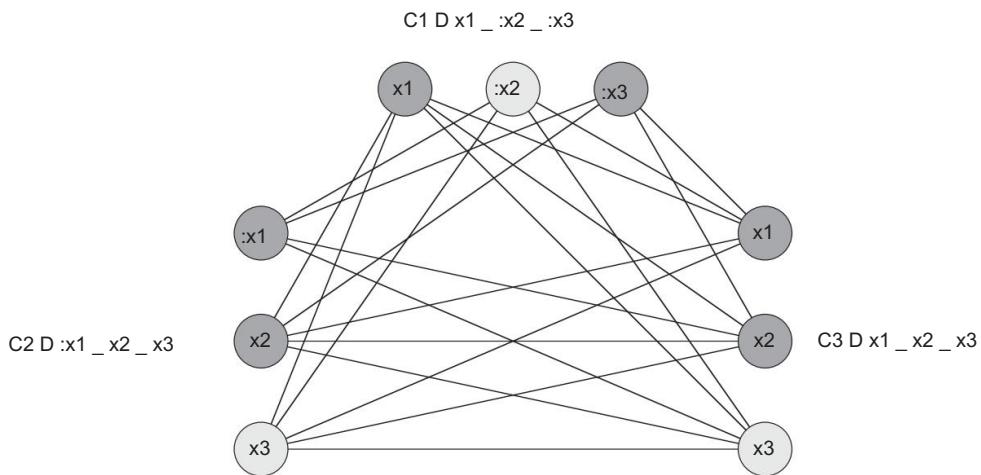


Figura 34.14 La gráfica  $G$  derivada de la fórmula 3-CNF  $D \ C1 \wedge C2 \wedge C3$ , donde  $C1 \ D \ .x1 \ _ \ :x2 \ _ \ :x3 /$ ,  $C2 \ D \ .:x1 \ _ \ x2 \ _ \ x3 /$ , y  $C3 \ D \ .x1 \ _ \ x2 \ _ \ x3 /$ , en la reducción de 3-CNF-SAT a CLIQUE. Una asignación satisfactoria de la fórmula tiene  $x2 \ D \ 0$ ,  $x3 \ D \ 1$  y  $x1 \ 0$  o  $1$ . Esta asignación satisface  $C1$  con  $:x2$ , y satisface  $C2$  y  $C3$  con  $x3$ , lo que corresponde a la camarilla con vértices ligeramente sombreados.

1; 2; ; ; ;  $k$ , cada cláusula  $C_r$  tiene exactamente tres literales distintos  $I_r$   
un grafo  $G$  tal que sea satisfacible si y solo si  $G$  tiene una camarilla de tamaño  $k$ .

Construimos el grafo  $GD$  . $V$ ;  $E$ / como sigue. Para cada cláusula  $C_r \ D \ .I_r \ _ \ I_r /$  en colocamos un triple de vértices  $r$  y  $r$  en  $\mathbb{V}$ . Ponemos un borde entre dos vértices  $r$  y  $s$  si ambos de los siguientes se cumplen: j

i

y  $s$  están en ternas diferentes, es decir,  $r \neq s$ , y j

sus literales correspondientes son consistentes, es decir,  $I_r$  no es la negación de  $I_s$

j.

Podemos construir fácilmente este gráfico en tiempo polinomial. Como ejemplo de esta construcción, si tenemos

$D \ .x1 \ _ \ :x2 \ _ \ :x3 / \wedge \ .:x1 \ _ \ x2 \ _ \ x3 / \wedge \ .x1 \ _ \ x2 \ _ \ x3 /;$

entonces  $G$  es la gráfica que se muestra en la figura 34.14.

Debemos demostrar que esta transformación de en  $G$  es una reducción. Primero, supongamos que tiene una tarea satisfactoria. Entonces, cada cláusula  $C_r$  contiene al menos un literal  $I_r$  al que se le asigna 1, y cada uno de esos literales corresponde a un vértice  $i$ . Selección de  $k$  vértices. Afirmamos que uno de esos literales "verdaderos" de cada cláusula produce un conjunto  $V^0$   $V^0$  camarilla. Para dos vértices cualquiera, los  $r$  y  $s$ ,  $j$ , donde  $r \neq s$ , ambos correspondientes son una literales  $r$  y  $s$  se asignan a 1 mediante la asignación satisfactoria dada y, por lo tanto, los literales

no pueden ser complementos. Así, por la construcción de  $G$ , la arista  $.r$  a  $E$ .  $i : j$   $s /$  pertenece

Por el contrario, supongamos que  $G$  tiene una camarilla<sup>0</sup> de tamaño  $k$ . Ningún borde en  $G$  connecta vértices en el mismo triple, por lo que  $V$ <sup>0</sup> contiene exactamente un vértice por triple. Podemos sin asignar 1 a cada literal  $| r$  tal que  $r$  i literal y su  $2^{\text{voltios}}{}^0$  temor a asignar 1 a ambos a complemento, ya que  $G$  no contiene bordes entre literales inconsistentes.

Cada cláusula se cumple, y así se cumple. (Cualquier variable que no corresponda a un vértice en la camarilla puede establecerse arbitrariamente). ■

En el ejemplo de la figura 34.14, una asignación satisfactoria de tiene  $x_2 = 0$  y  $x_3 = 1$ . Una camarilla correspondiente de tamaño  $k = 3$  consta de los vértices correspondientes a  $:x_2$  de la primera cláusula,  $x_3$  de la segunda cláusula y  $x_3$  de la cláusula tercera. Debido a que la camarilla no contiene vértices correspondientes a  $x_1$  o  $:x_1$ , podemos establecer  $x_1$  en 0 o 1 en esta asignación satisfactoria.

Observe que en la demostración del teorema 34.11, reducimos una instancia arbitraria de 3-CNF-SAT a una instancia de CLIQUE con una estructura particular. Podría pensar que solo hemos demostrado que CLIQUE es NP-duro en gráficos en los que los vértices están restringidos a ocurrir en triples y en los que no hay bordes entre los vértices en el mismo triple. De hecho, hemos demostrado que CLIQUE es NP-hard solo en este caso restringido, pero esta prueba es suficiente para mostrar que CLIQUE es NP-hard en gráficos generales. ¿Por qué? Si tuviéramos un algoritmo de tiempo polinomial que resolviera CLIQUE en gráficos generales, también resolvería CLIQUE en gráficos restringidos.

Sin embargo, el enfoque opuesto —reducir instancias de 3-CNF-SAT con una estructura especial a instancias generales de CLIQUE— no habría sido suficiente. ¿Por qué no? Quizás las instancias de 3-CNF-SAT que elegimos para reducir eran "fáciles", por lo que no habríamos reducido un problema NP-difícil a CLIQUE.

Observe también que la reducción utilizó la instancia de 3-CNF-SAT, pero no la solución. Habríamos errado si la reducción en tiempo polinomial se hubiera basado en saber si la fórmula es satisfecha, ya que no sabemos cómo decidir si es satisfecha en tiempo polinomial.

#### 34.5.2 El problema de la cobertura de vértices

Una cubierta de vértice de un grafo no dirigido  $G = (V, E)$  es un subconjunto  $U \subseteq V$  tal que si  $u \in U$ , luego  $u$  está conectado a todos los vértices en  $U$  (o ambos). Es decir, cada vértice "cubre" sus aristas incidentes, y una cubierta de vértice para  $G$  es un conjunto de vértices que cubre todas las aristas de  $E$ . El tamaño de una cubierta de vértice es el número de vértices en ella. Por ejemplo, el gráfico de la figura 34.15(b) tiene un vértice cubierto de tamaño 2.

El problema de la cobertura de vértices es encontrar una cubierta de vértices de tamaño mínimo en un gráfico dado. Replanteando este problema de optimización como un problema de decisión, deseamos

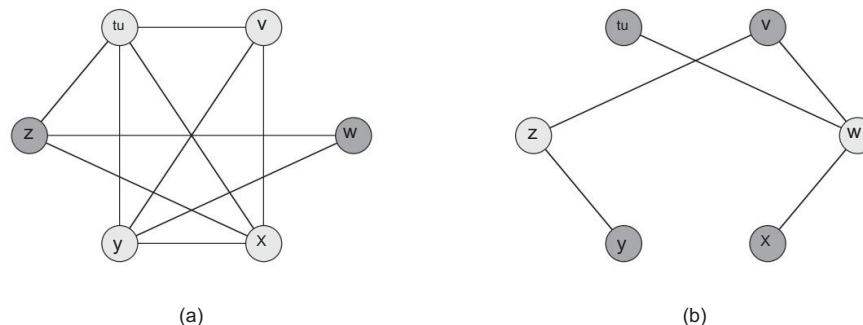


Figura 34.15 Reducción de CLIQUE a VERTEX-COVER. (a) Un grafo no dirigido GD .V; E/ con camarilla VD fu; ; X; yg. (b) El gráfico G producido por el algoritmo de reducción que tiene una cobertura de vértice D VV fw; gramo.

determinar si un gráfico tiene una cubierta de vértice de un tamaño  $k$  dado. Como lenguaje, nosotros definir

VERTEX-CUBIERTA D fHG; si W el gráfico G tiene una cubierta de vértice de tamaño kg :

El siguiente teorema muestra que este problema es NP-completo.

### Teorema 34.12 El

problema de cobertura de vértices es NP-completo.

Prueba Primero mostramos que VERTEX-COVER 2 NP. Supongamos que nos dan un grafo  $G = (V, E)$  y un entero  $k$ . El certificado que elegimos es la propia cubierta de vértice  $V$ . El algoritmo de verificación afirma que  $\sum_{v \in V} |N(v)| \geq 2k$ , que  $2k$  es polinomial en el tamaño del certificado. Podemos verificar fácilmente el certificado en  $O(2^{|V|} \cdot |E|)$ .

Probamos que el problema de la cobertura de vértices es NP-difícil mostrando que CLIQUE P VERTEX-COVER. Esta reducción se basa en la noción del "complemento" de un gráfico. Dado un grafo no dirigido  $G = \langle V; E \rangle$ , definimos el complemento de  $G$  como  $\bar{G} = \langle V; \bar{E} \rangle$ , donde  $\bar{E}$  es el conjunto de aristas que no están en  $E$ . La figura 34.15 muestra un grafo y su complemento e ilustra la reducción de CLIQUE a VERTEX-COVER.

El algoritmo de reducción toma como entrada una instancia  $hG$ ;  $ki$  del problema de la camarilla. Calcula el complemento  $G$ , que podemos hacer fácilmente en tiempo polinomial. La salida del algoritmo de reducción es la instancia  $hG$ ;  $jV$   $ki$  del problema de cobertura de vértices. Para completar la prueba, mostramos que esta transformación es de hecho una

reducción: el grafo  $G$  tiene una clique de tamaño  $k$  si y solo si el grafo  $G$  tiene una cubierta de vértice de tamaño  $jV^0 j k$ .

Supongamos que  $G$  tiene una camarilla  $V^0 \subseteq V$  con  $|V^0| = k$ . Decimos que  $V^0$  es una cubierta de vértice  $jV^0$  en  $G$ . Sea  $u; /$  sea cualquier arista en  $E$ . Entonces,  $u; / \in E$ , lo que implica que al menos uno de  $u$  o no pertenece a  $V^0$  ya que cada par de vértices en  $V^0$  está conectado por una arista de  $E$ . De manera equivalente, al menos uno de  $u$  o está en  $V^0$  lo que significa que la arista  $u; /$  está cubierta por  $V^0$  desde  $E$ , cada arista  $\in V^0$ . Dado que  $u; /$  fue elegido arbitrariamente de  $E$  está cubierta por un vértice en  $V^0$  que tiene tamaño  $jV^0 j k$ , forma  $V^0$ . Por lo tanto, el conjunto  $V^0$ , una cubierta de vértice para  $G$ .

Por el contrario, suponga que  $\bar{G}$  tiene una cubierta de vértice  $V^0 \subseteq V$ , donde  $|V^0| = k$ . Entonces, para todos  $u; / \in E$ , entonces  $u; / \in V^0$  ambos. El  $2V^0$  ustedes;  $2V^0$  contrapositivo de esta implicación es que para todo  $u; / \in E$ , si tienes  $62V^0$  y  $62V^0$  es  $2V^0$ ,  $V^0$  entonces  $u; / \in E$ . En otras palabras,  $V^0$  una camarilla, y tiene tamaño  $jV^0 j k$ . ■

Dado que VERTEX-COVER es NP-completo, no esperamos encontrar un algoritmo de tiempo polinomial para encontrar una cobertura de vértice de tamaño mínimo. Sin embargo, la sección 35.1 presenta un “algoritmo de aproximación” de tiempo polinomial que produce soluciones “aproximadas” para el problema de cobertura de vértices. El tamaño de una cubierta de vértice producida por el algoritmo es como mucho el doble del tamaño mínimo de una cubierta de vértice.

Por lo tanto, no debemos perder la esperanza solo porque un problema es NP-completo. Es posible que podamos diseñar un algoritmo de aproximación de tiempo polinomial que obtenga soluciones casi óptimas, aunque encontrar una solución óptima es NP-completo.

El Capítulo 35 proporciona varios algoritmos de aproximación para problemas NP-completos.

### 34.5.3 El problema del ciclo hamiltoniano

Ahora volvemos al problema del ciclo hamiltoniano definido en la Sección 34.2.

#### Teorema 34.13

El problema del ciclo hamiltoniano es NP-completo.

Prueba Primero mostramos que HAM-CYCLE pertenece a NP. Dado un grafo  $G = (V, E)$ , nuestro certificado es la secuencia de  $jV$  vértices que componen el ciclo hamiltoniano. El algoritmo de verificación comprueba que esta sucesión contiene cada vértice en  $V$  exactamente una vez y que con el primer vértice repetido al final forma un ciclo en  $G$ . Es decir, comprueba que existe una arista entre cada par de vértices consecutivos y entre el primer y último vértice. Podemos verificar el certificado en tiempo polinomial.

Ahora probamos que VERTEX-COVER P HAM-CYCLE, lo que demuestra que HAM-CYCLE es NP-completo. Dado un grafo no dirigido  $G = (V, E)$  y un

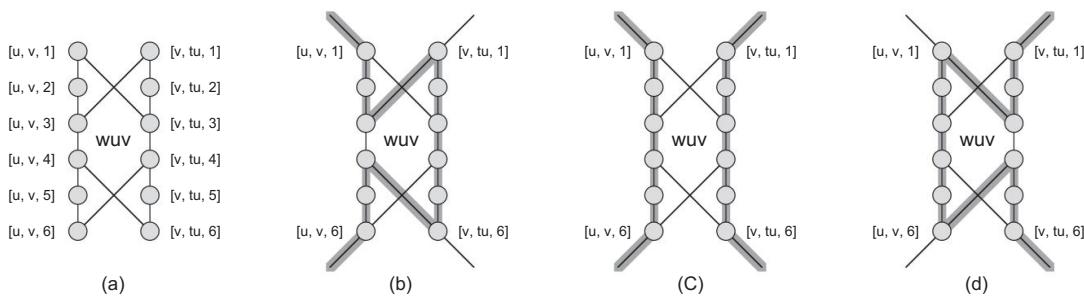


Figura 34.16 El widget utilizado para reducir el problema de cobertura de vértices al problema del ciclo hamiltoniano. Una arista  $.u; /$  del gráfico  $G$  corresponde al widget  $W_u$  en el gráfico  $G_0$  creado en la reducción. (a) El widget, con vértices individuales etiquetados. (b)–(d) Los caminos sombreados son los únicos posibles a través del widget que incluyen todos los vértices, asumiendo que las únicas conexiones desde el widget al resto de  $G_0$  son a través de los vértices  $\{u; ; 1, u; ; 6, O; tu; 1\}$  y  $\{u; ; 6\}$ .

entero  $k$ , construimos un grafo no dirigido  $G_0$  .  $V$  ciclo  $^0 : E_0 /$  que tiene un hamiltoniano si y solo si  $G$  tiene una cubierta de vértice de tamaño  $k$ .

Nuestra construcción utiliza un widget, que es una parte de un gráfico que impone ciertas propiedades. La figura 34.16(a) muestra el widget que usamos. Para cada arista  $.u; / 2 E$ , el gráfico  $G_0$  que construimos contendrá una copia de este widget, que denotaremos por  $W_u$ . Denotamos cada vértice en  $W_u$  por  $\{u; ; 1, u; ; 6, O; tu; 1\} \cup \{u; ; 6\}$ , donde  $1 \leq i \leq 6$ , por lo que cada widget  $W_u$  contiene 12 vértices. Widget  $W_u$  también contiene los 14 bordes que se muestran en la Figura 34.16(a).

Junto con la estructura interna del widget, aplicamos las propiedades que queremos al limitar las conexiones entre el widget y el resto del gráfico  $G_0$  que construimos. En particular, solo los vértices  $\{u; ; 1, u; ; 6, O; tu; 1\} \cup \{u; ; 6\}$  tendrán bordes incidentes desde fuera de  $W_u$ . Cualquier ciclo hamiltoniano de  $G_0$  debe atravesar las aristas de  $W_u$  en una de las tres formas que se muestran en las figuras 34.16(b)–(d). Si el ciclo entra por el vértice  $\{u; ; 1\}$ , debe salir por el vértice  $\{u; ; 6\}$  y visita los 12 vértices del widget (Figura 34.16(b)) o los seis vértices  $\{u; ; 1\} \cup \{u; ; 6\}$  (Figura 34.16(c)). En este último caso, el ciclo tendrá que volver a entrar en el widget para visitar los vértices  $\{u; ; 1\} \cup \{u; ; 6\}$ . Análogamente, si el ciclo entra por el vértice  $\{u; ; 6\}$ , debe salir por el vértice  $\{u; ; 1\}$  y visita los 12 vértices del widget (Figura 34.16(d)) o los seis vértices  $\{u; ; 1\} \cup \{u; ; 6\}$  (Figura 34.16(c)). No son posibles otras rutas a través del widget que visiten los 12 vértices. En particular, es imposible construir dos caminos disjuntos de vértice, uno de los cuales conecta  $\{u; ; 1\} \cup \{u; ; 6\}$  y el otro de los cuales conecta  $\{u; ; 1\} \cup \{u; ; 6\}$ , de modo que la unión de los dos caminos contenga todos los vértices del widget.

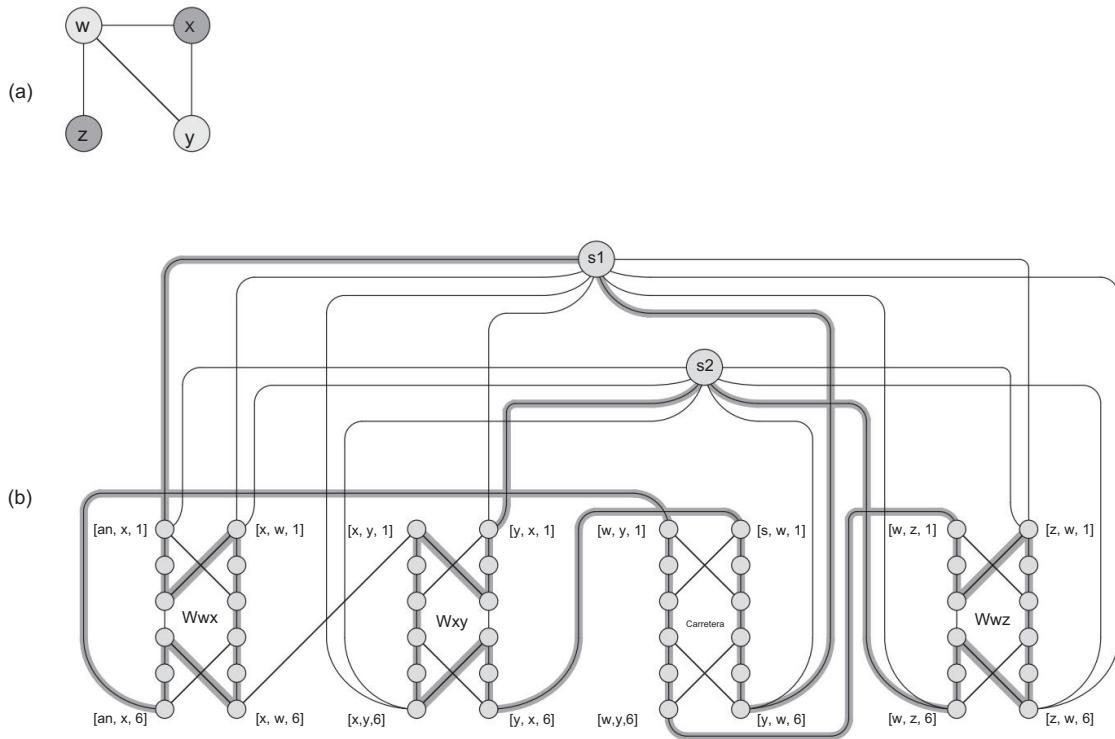


Figura 34.17 Reducción de una instancia del problema de cobertura de vértices a una instancia del problema del ciclo hamiltoniano. (a) Un grafo no dirigido  $G$  con una cubierta de vértices de tamaño 2, que consta de los vértices  $w$  y  $y$  ligeramente sombreados. (b) El gráfico no dirigido  $G_0$  producido por la reducción, con el camino hamiltoniano correspondiente a la cubierta de vértices sombreado. La cubierta de vértice  $fw; yg$  corresponde a las aristas  $.s1; \overrightarrow{ew}; X; 1/ y .s2; \overrightarrow{ey}; X; 1/$  que aparece en el ciclo hamiltoniano.

Los únicos otros vértices en  $V \setminus \{w, y\}$  distintos de los de los widgets son los vértices del selector  $s_1; s_2; \dots; s_k$ . Usamos aristas incidentes en los vértices del selector en  $G_0$  para seleccionar los  $k$  vértices de la cubierta en  $G$ .

Además de los bordes de los widgets,  $E_0$  contiene otros dos tipos de bordes, que muestra la figura 34.17. Primero, para cada vértice  $u \in V \setminus \{w, y\}$  agregamos aristas para unir pares de widgets para formar un camino que contenga todos los widgets correspondientes a las aristas incidentes en  $u$  en  $G$ . Ordenamos arbitrariamente los vértices adyacentes a cada vértice  $u \in V \setminus \{w, y\}$  como  $u.1 /; u.2 /; \dots; u.grado.u //$ , donde  $grado.u$  es el número de vértices adyacentes a  $u$ . Creamos un camino en  $G_0$  a través de todos los widgets correspondientes a las aristas que inciden en  $u$  sumando a  $E_0$  las aristas  $f.\overrightarrow{eu}; interfaz de usuario /; 6; \overrightarrow{eu}; u.iC1 /; 1/ W 1 i grado.u / 1g$ . En la figura 34.17, por ejemplo, ordenamos los vértices adyacentes a  $w$  como  $x; y; z$ , por lo que la gráfica  $G_0$  en la parte (b) de la figura incluye las aristas

. $\infty_w; X; 6; \infty_w; y; 1/y \infty_w; y; 6; \infty_w; '$ ; 1/. Para cada vértice  $u \in V$ , estos bordes en  $G_0$  completan una ruta que contiene todos los widgets correspondientes a los bordes que inciden en  $u$  en  $G$ .

La intuición detrás de estos bordes es que si elegimos un vértice  $u \in V$  en la cubierta de vértice de  $G$ , podemos construir un camino desde  $\infty_u; u.1/$ ; 1 a  $\infty_u; u.grado.u//; 6$  en  $G_0$  que "cubre" todos los widgets correspondientes a los bordes que inciden en  $u$ . Es decir, para cada uno de estos widgets, digamos  $W_{ui}$ , la ruta incluye los 12 vértices (si  $u$  está en la cubierta de vértices pero  $ui$  / no lo está) o solo los seis vértices  $\infty_u; interfaz de usuario /; 1; \infty_u; interfaz de usuario /; 2; \dots; \infty_u; interfaz de usuario /; 6$  (si tanto  $u$  como  $ui$  / están en la cubierta de vértice).

El último tipo de arista en  $E_0$  se une al primer vértice  $\infty_u; u.1/$ ; 1 y el último vértice  $\infty_u; u.grado.u//; 6$  de cada uno de estos caminos a cada uno de los vértices del selector. Es decir, incluimos los bordes

$f.sj ; \infty_u; u.1/; 1/W \cup 2V \cup 1/j \text{ kg} [ f.sj ; \infty_u;$

$u.grado.u//; 6/W \cup 2V \cup 1/j \text{ kg} :$

A continuación, mostramos que el tamaño de  $G_0$  es un polinomio del tamaño de  $G$  y, por lo tanto, podemos construir  $G_0$  en un polinomio de tiempo del tamaño de  $G$ . Los vértices de  $G_0$  son los de los widgets, más los vértices del selector. Con 12 vértices por widget, más  $k$  vértices selectores, tenemos un total de

$jV^0 j D 12 jEj C k 12 jEj$

$C jV j$

vértices. Los bordes de  $G_0$  son los de los widgets, los que van entre los widgets y los que conectan los vértices del selector a los widgets. Cada widget contiene 14 aristas, el gráfico totaliza  $14 jEj$  en todos los widgets. Para cada vértice  $u \in V$ ,  $G_0$  tiene un grado  $u/1$  que aristas que van entre los widgets, por lo que se suman todos los vértices en  $V$ .

$X.grado.u/1/D 2 jEj jV j$

$u2V$

los bordes van entre los widgets. Finalmente,  $G_0$  tiene dos aristas por cada par formado por un vértice selector y un vértice de  $V$ , totalizando  $2k jV j$  de dichas aristas. Por lo tanto, el número total de aristas de  $G_0$  es

$jE0 j D .14 jEj/C .2 jEj jV j/C .2k jV j/D 16 jEj C .2k 1/$

$jV j$

$16 jEj C .2 jV j 1/jV j$

Ahora mostramos que la transformación de la gráfica  $G$  a  $G_0$  es una reducción. Es decir, debemos demostrar que  $G$  tiene una cubierta de vértice de tamaño  $k$  si y solo si  $G_0$  tiene un ciclo hamiltoniano.

Supongamos que  $G = \langle V, E \rangle$  tiene una cubierta de vértice  $V$  de tamaño  $k$ . Dejar  $V = \{v_1, v_2, \dots, v_k\}$ . Como muestra la figura 34.17, formamos un ciclo hamiltoniano en  $G$  incluyendo las siguientes aristas: para cada vértice  $v_j \in V$ , incluye  $v_i / v_j C_1 / v_i$  bordes  $\{v_i, v_j\}$  para  $i \neq j$ ; y  $C_1 = \{v_1, v_2, \dots, v_k, v_1\}$ . También incluimos los bordes dentro de estos widgets como muestran las Figuras 34.16(b)–(d), dependiendo de si el borde está cubierto por uno o dos vértices en  $V$ . El ciclo hamiltoniano también incluye las aristas  $\{v_i, v_j\}$  para  $i \neq j$ .

f.sj ; ØEuj ; µ; ¼/ W 1 j kg u.grado.uj //  
ØEuj ; ; 6/ W 1 jk [f.sjC1;  
| f.s1; ØEuk; u.degree.uk// 6/g :

Al inspeccionar la figura 34.17, puede verificar que estos bordes forman un ciclo. El ciclo comienza en  $s_1$ , visita todos los widgets correspondientes a los bordes que inciden en  $u_1$ , luego visita  $s_2$ , visita todos los widgets correspondientes a los bordes que inciden en  $u_2$ , y así sucesivamente, hasta que regresa a  $s_1$ . El ciclo visita cada widget una o dos veces, dependiendo de si uno o dos vértices de  $V$  cubren su borde correspondiente. Debido a que  $V$  es una cubierta de vértice para  $G$ , cada borde en  $E$  incide en algún vértice en  $V$  y así el ciclo visita cada vértice en cada widget de  $G_0$ . Como el ciclo también visita todos los vértices del selector, es hamiltoniano.

Por el contrario, suponga que  $G \neq \emptyset$ :  $E_G$  / tiene un ciclo hamiltoniano  $C \in E_G$ .  
 afirmar que el conjunto

Nosotros

VD fu 2 VW .si : CEu: u.1/; 1/ 2 C por unos 1 i kg (34.4)

es una cubierta de vértice para G. Para ver por qué, divide C en caminos máximos que borde .si;  $\cap_{u \in U}$ ; u.1/ algún vértice selector si entra en algún u 2 V y finaliza en un vértice selector sj sin pasar por ningún otro vértice selector. Llamemos a cada uno de esos caminos un "camino de cobertura". Por cómo se construye  $G_0$ , cada ruta de cobertura debe tomar el si , todos los      borde .si ;  $\cap_{u \in U}$ ; u.1/ para algún vértice u 2 V , pasar por inicio en algún widgets correspondientes a aristas en E incidentes en u, y luego terminar en algún vértice selector sj . Nos referimos a este camino de cobertura como pu, y por la ecuación (34.4), ponemos u en V. Cada widget visitado por pu debe ser Wu o Wu por unos 2 V .

Para cada widget visitado por pu, sus vértices son visitados por uno o dos caminos de cobertura. Si son visitados por un camino de cobertura, entonces borde .u; / 2 E está cubierto en G por el vértice u. Si dos rutas de cobertura visitan el widget, entonces la otra ruta de cobertura debe ser p. lo que implica que borde .u; / 2 E está cubierto tanto por u como por

<sup>10</sup>Técnicamente, definimos un ciclo en términos de vértices en lugar de aristas (ver Sección B.4). En aras de la claridad, aquí abusamos de la notación y definimos el ciclo hamiltoniano en términos de aristas.

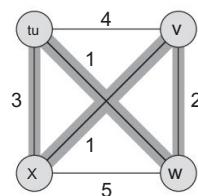


Figura 34.18 Ejemplo del problema del viajante de comercio. Los bordes sombreados representan un recorrido de costo mínimo, con un costo de 7.

Debido a que cada vértice en cada widget es visitado por alguna ruta de cobertura, vemos que cada borde en  $E$  está cubierto por algún vértice en  $V$ . ■

#### 34.5.4 El problema del viajante de comercio

En el problema del viajante de comercio, que está estrechamente relacionado con el problema del ciclo hamiltoniano, un vendedor debe visitar  $n$  ciudades. Modelando el problema como un grafo completo de  $n$  vértices, podemos decir que el vendedor desea hacer un recorrido, o ciclo hamiltoniano, visitando cada ciudad exactamente una vez y terminando en la ciudad de donde parte. El vendedor incurre en un costo entero no negativo  $c_{ij}$  para viajar desde la ciudad  $i$  y el donde el vendedor desea hacer el recorrido cuyo costo total es mínimo, hasta la ciudad  $j$ , costo total es la suma de los costos individuales a lo largo de los bordes del recorrido.

Por ejemplo, en la figura 34.18, un recorrido de costo mínimo es  $hu; w; ; X; ui$ , con costo 7.

El lenguaje formal para el problema de decisión correspondiente es

TSP D  $\{f, h, G\}; C; k \in \mathbb{N}$   
 $G$  es un grafo completo;  $C$  es una  
 función de  $V \times V \rightarrow \mathbb{Z}$ ;  $k \in \mathbb{Z}$ , y  $G$  tiene una  
 gira de  
 vendedor ambulante con costo máximo  $k$  :

El siguiente teorema muestra que es poco probable que exista un algoritmo rápido para el problema del viajante de comercio.

#### Teorema 34.14

El problema del viajante de comercio es NP-completo.

Prueba Primero mostramos que TSP pertenece a NP. Dada una instancia del problema, usamos como certificado la secuencia de  $n$  vértices en el recorrido. El algoritmo de verificación verifica que esta secuencia contenga cada vértice exactamente una vez, suma los costos de los bordes y verifica si la suma es como máximo  $k$ . Este proceso ciertamente se puede hacer en tiempo polinomial.

Para probar que TSP es NP-duro, mostramos que HAM-CYCLE TSP. Sea  
pág.  
 $G = \langle V, E \rangle$  ser una instancia de HAM-CYCLE. Construimos una instancia de TSP de la siguiente manera. Formamos el grafo completo  $G_0 = \langle V, E_0 \rangle$ , donde  $E_0 = \{(i, j) \in V^2 \mid i \neq j\}$ , y definimos la función de costo  $c$  por

$$c(i, j) = \begin{cases} 0 & \text{si } i = j \\ 1 & \text{si } i \neq j \end{cases}$$

(Observe que debido a que  $G$  no está dirigido, no tiene bucles propios, por lo que  $c(i, i) = 0$  para todos los vértices  $i \in V$ .) La instancia de TSP es entonces  $\langle G_0, c, 0 \rangle$ , que podemos crear fácilmente en tiempo polinomial.

Ahora mostramos que el grafo  $G$  tiene un ciclo hamiltoniano si y solo si el grafo  $G_0$  tiene un recorrido de costo a lo sumo 0. Supongamos que el grafo  $G$  tiene un ciclo hamiltoniano  $h$ . Cada arista en  $h$  pertenece a  $E$  y por lo tanto tiene un costo de 0 en  $G_0$ . Entonces,  $h$  es un recorrido en  $G_0$  con costo 0. Por el contrario, supongamos que el gráfico  $G_0$  tiene un recorrido  $h_0$  de costo máximo 0. Como los costos de las aristas en  $E_0$  son 0 y 1, el costo del recorrido  $h_0$  es exactamente 0 y cada arista del recorrido debe tener un costo de 0. Por lo tanto,  $h_0$  contiene solo aristas en  $E$ . Concluimos que  $h_0$  es un ciclo hamiltoniano en el gráfico  $G$ . ■

#### 34.5.5 El problema de la suma de subconjuntos

A continuación, consideraremos un problema aritmético NP-completo. En el problema de suma de subconjuntos, tenemos un conjunto finito  $S$  de enteros positivos y un objetivo de entero  $t > 0$ . Preguntamos si existe un subconjunto  $S_0 \subseteq S$  cuyos elementos suman  $t$ . Por ejemplo, si  $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$  y  $t = 138457$ , luego el subconjunto  $S_0 = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 17206, 117705\}$  es una solución.

Como de costumbre, definimos el problema como un lenguaje:

$\text{SUBSET-SUM} = \{ \langle S, t \rangle \mid \text{existe un subconjunto } S_0 \subseteq S \text{ tal que } \sum_{i \in S_0} i = t \}$

$\leq_s$

Como con cualquier problema aritmético, es importante recordar que nuestra codificación estándar asume que los enteros de entrada están codificados en binario. Con esta suposición en mente, podemos mostrar que es poco probable que el problema de suma de subconjuntos tenga un algoritmo rápido.

#### Teorema 34.15 El

problema de suma de subconjuntos es NP-completo.

Prueba Para mostrar que SUBSET-SUM está en NP, para una instancia  $\langle S, t \rangle$  del problema, dejemos que el subconjunto  $S_0$  sea el certificado. Un algoritmo de verificación puede verificar si  $\sum_{i \in S_0} i = t$ .

$\leq_s$

Ahora mostramos que  $\text{3-CNF-SAT} \leq_s \text{SUBCONJUNTO-SUMA}$ . Dada una fórmula 3-CNF

sobre variables  $x_1, x_2, \dots, x_n$  con cláusulas  $C_1, C_2, \dots, C_k$ , cada uno conteniendo exactamente

tres literales distintos, el algoritmo de reducción construye una instancia hS; t del problema de suma de subconjuntos tal que es satisfecha si y solo si existe un subconjunto de S cuya suma es exactamente t. Sin pérdida de generalidad, hacemos dos supuestos simplificadores sobre la fórmula. En primer lugar, ninguna cláusula contiene tanto una variable como su negación, pues tal cláusula se satisface automáticamente mediante cualquier asignación de valores a las variables. En segundo lugar, cada variable aparece en al menos una cláusula, porque no importa qué valor se asigne a una variable que no aparece en ninguna cláusula.

La reducción crea dos números en el conjunto S para cada variable  $x_i$  y dos números en S para cada cláusula  $C_j$ . Crearemos números en base 10, donde cada número contiene  $nC_k$  dígitos y cada dígito corresponde a una variable o una cláusula.

La base 10 (y otras bases, como veremos) tiene la propiedad que necesitamos de evitar acarreos de dígitos inferiores a dígitos superiores.

Como muestra la figura 34.19, construimos el conjunto S y el objetivo t de la siguiente manera. Etiquetamos cada posición de dígito por una variable o una cláusula. Los k dígitos menos significativos están etiquetados por las cláusulas, y los n dígitos más significativos están etiquetados por variables.

El objetivo t tiene un 1 en cada dígito etiquetado por una variable y un 4 en cada dígito etiquetado por una cláusula.

Para cada variable  $x_i$ , el conjunto S contiene dos enteros  $s_i^0$  y  $s_i^1$ . Cada uno de  $s_i^0$  y  $s_i^1$  tiene un 1 en el dígito etiquetado por  $x_i$  y 0 en los otros dígitos de las variables. Si el literal  $x_i$  aparece en la cláusula  $C_j$ , entonces el dígito etiquetado por  $C_j$  en  $s_i^0$  contiene un 1. Si el literal  $\neg x_i$  aparece en la cláusula  $C_j$ , entonces el dígito etiquetado por  $C_j$  en  $s_i^1$  contiene un 1. Todos los demás dígitos etiquetados por cláusulas en y son 0.

Todos los valores pueden igualarse en los dígitos más significativos. Además, mediante nuestra simplificación puede ser igual en todos los k dígitos menos significativos. Supuestos y  $\neg x_i$  tendrían que aparecer anteriores, no  $i$  y  $\neg i$  fueran iguales, entonces  $x_i$  exactamente en el mismo conjunto de cláusulas. Pero asumimos que ninguna cláusula contiene tanto  $x_i$  como  $\neg x_i$  o que  $x_i$  o  $\neg x_i$  aparecen en alguna cláusula, por lo que debe haber alguna cláusula  $C_j$  para la cual Para cada cláusula  $C_j$ , 0s en todos los dígitos excepto el etiquetado por  $C_j$ . y  $\neg x_i$  no difieren de.

Para  $s_j$ , digit y  $s_0$  use el conjunto S contiene dos enteros  $s_j^0$  y  $s_j^1$ . Cada uno de  $s_j^0$  y  $s_j^1$  tiene para obtener cada posición de dígito etiquetada con cláusula para  $s_0$  tiene un 1 en el  $C_j$  agregar al  $s_j^1$  tiene un 2 en este dígito. Estos enteros son "variables de holgura", que valor objetivo de 4.

La simple inspección de la figura 34.19 demuestra que todos los valores de  $s_j^0$  y  $s_j^1$  en S son únicos en el conjunto S.

Tenga en cuenta que la mayor suma de dígitos en cualquier posición de un dígito es 6, que ocurre en los dígitos etiquetados por cláusulas (tres 1 de los valores, más 1 y 2 de

	x1	x2	x3	C1	C2	C3	C4
1	=1001001						
<sub>01</sub>	=1000110						
2	=0100001						
<sub>02</sub>	=0101110						
3	=0010011						
<sub>03</sub>	=0011100						
s1	=0001000						
<sub>s01</sub>	=0002000						
s2	=0000100						
<sub>s02</sub>	=0000200						
s3	=0000010						
<sub>s03</sub>	=0000020						
s4	=0000001						
<sub>s04</sub>	=0000002						
t	=1114444						

Figura 34.19 La reducción de 3-CNF-SAT a SUBSET-SUM. La fórmula en 3-CNF es  $D \cdot C1 \wedge C2 \wedge C3 \wedge C4$ , donde  $C1 \cdot D \cdot x1 \cdot x2 \cdot x3$ ,  $C2 \cdot D \cdot x1 \cdot x2 \cdot x3$ ,  $C3 \cdot D \cdot x1 \cdot x2 \cdot x3$  y  $C4 \cdot D \cdot x1 \cdot x2 \cdot x3$ . Una asignación satisfactoria de es  $hx1 \cdot D \cdot 0$ ;  $x2 \cdot D \cdot 0$ ;  $x3 \cdot D \cdot 1$ . El conjunto S producido por la reducción consta de los números de base 10 que se muestran; lectura de arriba a abajo, SD f1001001; 1000110; 100001; 100110; 11100; 1000; 2000; 100; 200; 10; 20; 1; 2g El objetivo t es 1114444. El subconjunto  $S_0$  S está ligeramente sombreado y contiene y 3, correspondiente a la asignación satisfactoria. También contiene variables de  $h$  figura s1, s0 s0 2, s3, s4 y s0 para lograr el valor objetivo de 4 en los dígitos etiquetados de C1 a C4.

4

$s_j$  y  $s_0$  pueden valores). Interpretando estos números en base 10, por lo tanto, no hay acarreos. ocurrir de dígitos más bajos a dígitos más altos.<sup>11</sup>

Podemos realizar la reducción en tiempo polinomial . El conjunto S contiene  $2n$  C  $2k$  valores, cada uno de los cuales tiene  $n$  C  $k$  dígitos, y el tiempo para producir cada dígito es un polinomio en  $n$  C  $k$ . El objetivo t tiene  $n$  C  $k$  dígitos, y la reducción produce cada uno en tiempo constante.

Ahora mostramos que la fórmula 3-CNF es satisfactoria si y solo si existe un subconjunto  $S_0$  S cuya suma es t. Primero, supongamos que tiene una tarea satisfactoria.

Para  $i \in 1; 2; \dots; n$ , si  $x_i \cdot D \cdot 1$  en esta asignación, entonces incluir en  $S_0$  . De lo contrario, incluir En otras palabras, incluimos en  $S_0$  exactamente los valores y que  $c_{0i}$

---

<sup>11</sup>De hecho, cualquier base b, donde b 7, funcionaría. La instancia al comienzo de esta subsección es el conjunto S y el objetivo t en la figura 34.19 interpretados en base 7, con S enumerados en orden.

responder a los literales con el valor 1 en la asignación satisfactoria. Habiendo incluido cualquiera pero no ambos, para todos los  $i$ , y habiendo puesto 0 en los dígitos etiquetados por  $i$  o variables en ~~S0 de los que se cumplen~~ para cada dígito etiquetado por variable, la suma de los valores  $j$ , de con esos dígitos del objetivo  $t$ . Debido a que cada cláusula se cumple, la cláusula contiene algún literal con el valor 1. Por lo tanto, cada dígito etiquetado por una cláusula tiene al menos un 1 contribuido a su suma por un  $i$  o valor en  $S_0$ .

De hecho, 1, 2 o 3 literales pueden ser 1 en cada cláusula, por lo que cada cláusula en  $S_0$ . El dígito etiquetado tiene una suma de 1, 2 o 3 de y En la Figura 34.18 i , por ejemplo, los literales : $x_1$ , : $x_2$  y  $x_3$  tienen el valor 1 en una asignación satisfactoria.

Cada una de las cláusulas  $C_1$  y  $C_4$  contiene exactamente uno de estos literales, y así juntos  $x_2$ , y  $x_3$  contribuye con 1 a la suma de los dígitos de  $C_1$  y  $C_4$ . La cláusula  $C_2$  contiene dos de estos literales, y y La cláusula  $C_3$  contribuir 2 a la suma en el dígito de  $C_2$ . y aporta contiene estos tres literales, y suma en el dígito para  $C_3$ .  $x_1$ ,  $x_2$ , 3 a la  $x_3$

Alcanzamos el objetivo de 4 en cada dígito etiquetado por la cláusula  $C_j$  al incluir en  $S_0$  el subconjunto no vacío apropiado de variables de holgura  $s_j$ ;  $s_0$  g. En la Figura 34.19,  $S_0$  incluye  $s_1$ ,  $s_0$  2,  $s_3$ ,  $s_4$  y  $s_0$  Como hemos igualado el objetivo en todos los dígitos de la suma y no pueden ocurrir acarreos, los valores de  $S_0$  suman  $t$ .

Ahora, suponga que hay un subconjunto  $S_0$  S que suma  $t$ . El subconjunto  $S_0$  debe incluir exactamente uno de para cada  $j$  D 1; 2; : : ; n, porque de lo contrario los dígitos etiquetados por variables no sumarían 1. Si establecemos  $x_i$  D 1. De lo contrario, establecemos  $x_i$  D 0. Decimos  $x_i$  2  $S_0$ , que cada cláusula  $C_j$ , para  $j$  D 1; 2; : : ; k, está satisfecho con esta asignación. Para probar esta afirmación, tenga en cuenta que para lograr una suma de 4 en el dígito etiquetado por  $C_j$ , el subconjunto  $S_0$  debe incluir al menos un  $i$  o valor que tenga un 1 en el dígito etiquetado por  $C_j$ , ya que las contribuciones de las variables de holgura  $s_j$  y  $s_0$  que tiene un 1 en la posición de  $C_j$ ,  $j$  2  $S_0$ ntos suman como máximo 3. Si  $S_0$  incluye una  $i$ , entonces el literal  $x_i$  aparece en la cláusula  $C_j$ . Dado que hemos establecido  $x_i$  D 1 cuando se cumple la cláusula  $C_j$ . Si  $S_0$  incluye a que tiene un 1 en esa posición, entonces el literal : $x_i$  aparece en  $C_j$ . Dado que hemos establecido  $x_i$  D 0 cuando la cláusula  $C_j$  se cumple nuevamente. Por lo tanto, todas las cláusulas de están satisfechas, lo que completa la prueba. ■

## Ejercicios

### 34.5-1

El problema de isomorfismo de subgrafos toma dos grafos no dirigidos  $G_1$  y  $G_2$ , y pregunta si  $G_1$  es isomorfo a un subgrafo de  $G_2$ . Demuestre que el problema de isomorfismo de subgrafos es NP-completo.

### 34.5-2

Dada una matriz  $A$  entera  $mn$  y un vector  $b$  entero  $m$ , el problema de programación entera 0-1 pregunta si existe un  $n$ -vector  $x$  entero con elementos

mentos en el conjunto  $f_0$ ;  $1g$  tal que  $Ax \leq b$ . Demuestre que la programación entera 0-1 es NP-completa. (Sugerencia: reduzca de 3-CNF-SAT.)

#### 34.5-3

El problema de programación lineal con enteros es como el problema de programación con enteros 0-1 dado en el ejercicio 34.5-2, excepto que los valores del vector  $x$  pueden ser enteros en lugar de 0 o 1. El problema de programación es NP-difícil, demuestre que el problema de programación lineal con enteros es NP completo.

#### 34.5-4

Muestre cómo resolver el problema de suma de subconjuntos en tiempo polinomial si el valor objetivo  $t$  se expresa en unaryo.

#### 34.5-5

El problema de partición de conjuntos toma como entrada un conjunto  $S$  de números. La pregunta es si los números se pueden dividir en dos conjuntos  $A$  y  $B$  de manera que  $\sum_{x \in A} x = \sum_{x \in B}$ . Demuestre que el problema de partición de conjuntos es NP-completo.

#### 34.5-6

Muestre que el problema de la trayectoria hamiltoniana es NP-completo.

#### 34.5-7

El problema del ciclo simple más largo es el problema de determinar un ciclo simple (sin vértices repetidos) de longitud máxima en un gráfico. Formule un problema de decisión relacionado y demuestre que el problema de decisión es NP-completo.

#### 34.5-8

En el problema de satisfacibilidad de la mitad de 3-CNF, se nos da una fórmula de 3-CNF con  $n$  variables y  $m$  cláusulas, donde  $m$  es par. Deseamos determinar si existe una asignación de verdad a las variables de tal que exactamente la mitad de las cláusulas evalúen a 0 y exactamente la mitad de las cláusulas evalúen a 1. Demuestre que la mitad del problema de satisfacibilidad de 3-CNF es NP-completo.

## Problemas

### 34-1 Conjunto independiente

Conjunto independiente de un gráfico  $G = (V, E)$  es un subconjunto  $V$  en  $V$  de vértices tales . El problema es encontrar un conjunto independiente de tamaño máximo en  $G$ .

- a. Formule un problema de decisión relacionado para el problema de conjuntos independientes y demuestre que es NP-completo. (Sugerencia: reduzca del problema de la camarilla).
- b. Suponga que le dan una subrutina de “caja negra” para resolver el problema de decisión que definió en la parte (a). Dé un algoritmo para encontrar un conjunto independiente de tamaño máximo. El tiempo de ejecución de su algoritmo debe ser polinomial en  $jV$  y  $jEj$ , contando las consultas a la caja negra como un solo paso.

Aunque el problema de decisión de conjuntos independientes es NP-completo, ciertos casos especiales son resolvibles en tiempo polinomial.

- C. Proporcione un algoritmo eficiente para resolver el problema de conjuntos independientes cuando cada vértice en  $G$  tiene grado 2. Analice el tiempo de ejecución y demuestre que su algoritmo funciona correctamente.
- d. Proporcione un algoritmo eficiente para resolver el problema de conjuntos independientes cuando  $G$  es bipartito. Analice el tiempo de ejecución y demuestre que su algoritmo funciona correctamente. (Sugerencia: use los resultados de la Sección 26.3.)

#### 34-2 Bonnie y Clyde Bonnie y

Clyde acaban de robar un banco. Tienen una bolsa de dinero y quieren dividirla. Para cada uno de los siguientes escenarios, proporcione un algoritmo de tiempo polinomial o demuestre que el problema es NP-completo. La entrada en cada caso es una lista de los  $n$  elementos de la bolsa, junto con el valor de cada uno.

- a. La bolsa contiene  $n$  monedas, pero solo 2 denominaciones diferentes: algunas monedas valen  $x$  dólares y otras valen  $y$  dólares. Bonnie y Clyde desean dividir el dinero en partes exactamente iguales.
- b. La bolsa contiene  $n$  monedas, con un número arbitrario de denominaciones diferentes, pero cada denominación es una potencia entera no negativa de 2, es decir, las denominaciones posibles son 1 dólar, 2 dólares, 4 dólares, etc. Bonnie y Clyde desean dividir el dinero exactamente uniformemente.
- C. La bolsa contiene  $n$  cheques que, en una asombrosa coincidencia, están a nombre de "Bonnie o Clyde". Quieren dividir los cheques para que cada uno reciba exactamente la misma cantidad de dinero.
- d. La bolsa contiene  $n$  cheques como en el inciso (c), pero esta vez Bonnie y Clyde están dispuestos a aceptar una división en la que la diferencia no supere los 100 dólares.

### 34-3 Coloreado de gráficos

Los cartógrafos intentan usar la menor cantidad de colores posible cuando colorean países en un mapa, siempre que no haya dos países que comparten una frontera que tengan el mismo color.

Podemos modelar este problema con un grafo no dirigido  $G = (V, E)$  en la que cada vértice representa un país y los vértices cuyos respectivos países comparten una frontera son adyacentes.

Entonces, una coloración  $k$  es una función  $c: V \rightarrow \{1, 2, \dots, k\}$  tales que  $c(u) \neq c(v)$  para cada arista  $(u, v) \in E$ . En otras palabras, los números  $1, 2, \dots, k$  representan los  $k$  colores, y los vértices adyacentes deben tener colores diferentes. El problema de coloreado de gráficos consiste en determinar el número mínimo de colores necesarios para colorear un gráfico dado.

- a. Proporcione un algoritmo eficiente para determinar una coloración de 2 de un gráfico, si existe.
- b. Plantee el problema de coloración de gráficos como un problema de decisión. Demuestra que tu problema de decisión se puede resolver en tiempo polinomial si y solo si el problema de coloreado de gráficas se puede resolver en tiempo polinomial.
- c. Sea el lenguaje 3-COLOR el conjunto de gráficas que pueden ser de 3 colores. Demuestra que si 3-COLOR es NP-completo, entonces tu problema de decisión del inciso (b) es NP-completo.

Para probar que 3-COLOR es NP-completo, usamos una reducción de 3-CNF-SAT.

Dada una fórmula de  $m$  cláusulas sobre  $n$  variables  $x_1, x_2, \dots, x_n$ , construimos un grafo  $G = (V, E)$  como sigue. El conjunto  $V$  consta de un vértice para cada variable, un vértice para la negación de cada variable, 5 vértices para cada cláusula y 3 vértices especiales: VERDADERO, FALSO y ROJO. Los bordes del gráfico son de dos tipos: bordes "literales" que son independientes de las cláusulas y bordes "cláusula" que dependen de las cláusulas.

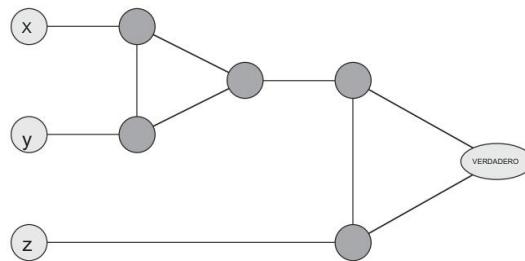
Los bordes literales forman un triángulo en los vértices especiales y también forman un triángulo en  $x_i, \neg x_i$  y RED para  $i \in \{1, 2, \dots, m\}$ .

- d. Argumente que en cualquier  $c$  de 3 colores de un gráfico que contiene los bordes literales, exactamente uno de una variable y su negación se colorea c.VERDADERO/ y el otro se colorea c.FALSO/. Argumente que para cualquier asignación de verdad existe una coloración de 3 colores del gráfico que contiene solo los bordes literales.

El widget que se muestra en la Figura 34.20 ayuda a hacer cumplir la condición correspondiente a una cláusula  $x_i \vee \neg x_i \vee y_j \vee \neg y_j$ . Cada cláusula requiere una copia única de los 5 vértices que están muy sombreados en la figura; se conectan como se muestra a los literales de la cláusula y el vértice especial VERDADERO.

- e. Argumente que si cada uno de  $x_i, \neg x_i, y_j, \neg y_j$  tiene el color c.VERDADERO/ o c.FALSO/, entonces el widget se puede colorear en 3 si y solo si al menos uno de  $x_i, \neg x_i, y_j, \neg y_j$  tiene el color c.VERDADERO/ .

- f. Complete la prueba de que 3-COLOR es NP-completo.

Figura 34.20 El widget correspondiente a una cláusula  $.x \_ y \_ '/$ , utilizado en el problema 34-3.

**34-4 Planificación con beneficios y plazos** Supongamos que tenemos una máquina y un conjunto de  $n$  tareas  $a_1, a_2, \dots, a_n$ , cada uno de los cuales requiere tiempo en la máquina. Cada tarea  $a_j$  requiere  $t_{ij}$  unidades de tiempo en el y tiene una fecha límite  $d_j$ . La máquina (su tiempo de procesamiento) produce una sola tarea a la vez y la tarea  $a_j$  debe ejecutarse sin interrupción durante  $t_{ij}$  ganancia de  $p_j$ , la máquina puede procesar unidades de tiempo consecutivas. Si completamos la tarea  $a_j$  antes de su fecha límite  $d_j$ , pero si la completamos después de su fecha límite, no recibimos ningún beneficio. Como recibir una ganancia  $p_j$ , un problema de optimización, nos deseamos encontrar  $\dots$  dan los tiempos de procesamiento, las ganancias y los plazos para un conjunto de  $n$  tareas, y programar que complete todas las tareas y devuelva la mayor cantidad de ganancias. Los tiempos de procesamiento, las ganancias y los plazos son todos números no negativos.

- Plantee este problema como un problema de decisión.
  - Demuestre que el problema de decisión es NP-completo.
- C. Proporcione un algoritmo de tiempo polinomial para el problema de decisión, suponiendo que todos los tiempos de procesamiento son números enteros de 1 a  $n$ . (Sugerencia: use programación dinámica).
- Dé un algoritmo de tiempo polinomial para el problema de optimización, suponiendo que todos los tiempos de procesamiento son números enteros de 1 a  $n$ .

### Notas del capítulo

El libro de Garey y Johnson [129] proporciona una guía maravillosa para la complejidad de NP, analiza la teoría en profundidad y proporciona un catálogo de muchos problemas que se sabía que eran NP-completos en 1979. La demostración del teorema 34.13 está adaptada de su libro, y la lista de dominios de problemas NP-completos al comienzo de la Sección 34.5 se extrae de su índice. Johnson escribió una serie

de 23 columnas en el *Journal of Algorithms* entre 1981 y 1992 informando nuevos desarrollos en NP-completitud. Hopcroft, Motwani y Ullman [177], Lewis y Papadimitriou [236], Papadimitriou [270] y Sipser [317] tienen buenos tratamientos de la completitud de NP en el contexto de la teoría de la complejidad. El NP-completo y varias reducciones también aparecen en libros de Aho, Hopcroft y Ullman [5]; Dasgupta, Papadimitriou y Vazirani [82]; Johnsonbaugh y Schaefer [193]; y Kleinberg y Tardos [208].

La clase P fue introducida en 1964 por Cobham [72] e, independientemente, en 1965 por Edmonds [100], quien también introdujo la clase NP y conjeturó que  $P \neq NP$ .

La noción de completitud de NP fue propuesta en 1971 por Cook [75], quien dio las primeras pruebas de completitud de NP para la satisfacibilidad de fórmulas y la satisfacibilidad de 3-CNF. Levin [234] descubrió la noción de forma independiente, dando una prueba de completitud NP para un problema de mosaico. Karp [199] introdujo la metodología de reducciones en 1972 y demostró la rica variedad de problemas NP-completos. El artículo de Karp incluía las pruebas originales de completitud NP de los problemas de la camarilla, la cobertura de vértices y el ciclo hamiltoniano. Desde entonces, muchos investigadores han demostrado que miles de problemas son NP-completos. En una charla en una reunión que celebraba el 60 cumpleaños de Karp en 1995, Papadimitriou comentó: "alrededor de 6000 artículos cada año tienen el término 'NP-completo' en su título, resumen o lista de palabras clave. Esto es más que cada uno de los términos 'compilador', 'base de datos', 'experto', 'red neuronal' o 'sistema operativo'."

El trabajo reciente en la teoría de la complejidad ha arrojado luz sobre la complejidad de calcular soluciones aproximadas. Este trabajo da una nueva definición de NP usando "pruebas comprobables probabilísticamente". Esta nueva definición implica que para problemas como la camarilla, la cobertura de vértices, el problema del vendedor ambulante con la desigualdad del triángulo y muchos otros, calcular buenas soluciones aproximadas es NP-difícil y, por lo tanto, no es más fácil que calcular soluciones óptimas. Una introducción a esta área puede encontrarse en la tesis de Arora [20]; un capítulo de Arora y Lund en Hochbaum [172]; un artículo de encuesta de Arora [21]; un libro editado por Mayr, Prömel y Steger [246]; y un artículo de encuesta de Johnson [191].

## 35 Algoritmos de aproximación

Muchos problemas de importancia práctica son NP-completos, pero son demasiado importantes para abandonarlos simplemente porque no sabemos cómo encontrar una solución óptima en tiempo polinomial. Incluso si un problema es NP-completo, puede haber esperanza. Tenemos al menos tres formas de sortear la completitud de NP. Primero, si las entradas reales son pequeñas, un algoritmo con tiempo de ejecución exponencial puede ser perfectamente satisfactorio. En segundo lugar, podemos aislar casos especiales importantes que podemos resolver en tiempo polinomial. En tercer lugar, podríamos encontrar enfoques para encontrar soluciones casi óptimas en tiempo polinomial (ya sea en el peor de los casos o en el caso esperado). En la práctica, casi lo óptimo suele ser lo suficientemente bueno. A un algoritmo que devuelve soluciones casi óptimas lo llamamos algoritmo de aproximación. Este capítulo presenta algoritmos de aproximación en tiempo polinomial para varios problemas NP-completos.

### Razones de rendimiento para algoritmos de aproximación

Supongamos que estamos trabajando en un problema de optimización en el que cada solución potencial tiene un costo positivo y deseamos encontrar una solución casi óptima. Dependiendo del problema, podemos definir una solución óptima como una con el máximo costo posible o una con el mínimo costo posible; es decir, el problema puede ser un problema de maximización o de minimización.

Decimos que un algoritmo para un problema tiene una relación de aproximación de  $\frac{C}{n}$  si, para cualquier entrada de tamaño  $n$ , el costo  $C$  de la solución producida por el algoritmo está dentro de un factor de  $\frac{C}{n}$  del costo  $C^*$  de una solución óptima. Solución:

$$\text{máximo } \frac{C}{C^*} : \frac{C}{n} : \quad (35.1)$$

Si un algoritmo logra una relación de aproximación de  $\frac{C}{n}$ , lo llamamos algoritmo de aproximación  $\frac{C}{n}$ . Las definiciones de la relación de aproximación y de un algoritmo de aproximación  $\frac{C}{n}$  se aplican tanto a problemas de minimización como de maximización. Para un problema de maximización,  $0 < C^* < C$ , y la razón  $C = C^*$  da el factor por el cual el costo de una solución óptima es mayor que el costo de la solución aproximada.

solución. De manera similar, para un problema de minimización,  $0 < CC$ , y la relación  $C=C$  da el factor por el cual el costo de la solución aproximada es mayor que el costo de una solución óptima. Debido a que suponemos que todas las soluciones tienen un costo positivo, estas razones siempre están bien definidas. La relación de aproximación de un algoritmo de aproximación nunca es menor que 1, ya que  $C=C$  1 implica  $C=C$  1.

Por lo tanto, un algoritmo de aproximación 1 produce una solución óptima, y un algoritmo de aproximación con una relación de aproximación grande puede devolver una solución que es mucho peor que la óptima.

Para muchos problemas, tenemos algoritmos de aproximación en tiempo polinomial con relaciones de aproximación constantes pequeñas, aunque para otros problemas, los algoritmos de aproximación en tiempo polinomial más conocidos tienen relaciones de aproximación que crecen en función del tamaño de entrada n. Un ejemplo de tal problema es el problema de cobertura del conjunto presentado en la Sección 35.3.

Algunos problemas NP-completos permiten algoritmos de aproximación de tiempo polinomial que pueden lograr proporciones de aproximación cada vez mejores utilizando más y más tiempo de cálculo. Es decir, podemos cambiar el tiempo de cálculo por la calidad de la aproximación. Un ejemplo es el problema de suma de subconjuntos estudiado en la Sección 35.5.

Esta situación es lo suficientemente importante como para merecer un nombre propio.

Un esquema de aproximación para un problema de optimización es un algoritmo de aproximación que toma como entrada no solo una instancia del problema, sino también un valor. El esquema es un algoritmo de  $>0$  tal que para cualquier fijo , aproximación de .1 C /.

Decimos que un esquema de aproximación es un esquema de aproximación de tiempo polinomial si para cualquier  $>0$  fijo, el esquema se ejecuta en polinomio de tiempo en el tamaño n de su instancia de entrada.

El tiempo de ejecución de un esquema de aproximación de tiempo polinomial puede aumentar muy rápidamente a medida que disminuye. Por ejemplo, el tiempo de ejecución de un esquema de aproximación de tiempo polinomial podría ser  $O.n^2=$ . Idealmente, si disminuye por un factor constante, el tiempo de ejecución para lograr la aproximación deseada no debería aumentar por más de un factor constante (aunque no necesariamente el mismo factor constante por el cual disminuyó).

Decimos que un esquema de aproximación es un esquema de aproximación de tiempo completamente polinomial si es un esquema de aproximación y su tiempo de ejecución es polinomial tanto en 1= como en el tamaño n de la instancia de entrada. Por ejemplo, el esquema podría tener un tiempo de ejecución de  $O..1=/2n^3=$ . Con tal esquema, cualquier disminución de factor constante viene con un aumento de factor constante correspondiente en el tiempo de ejecución.

<sup>1</sup>Cuando la relación de aproximación es independiente de n, usamos los términos "relación de aproximación de" y "algoritmo de aproximación", lo que indica que no depende de n.

### Bosquejo del capítulo

Las primeras cuatro secciones de este capítulo presentan algunos ejemplos de algoritmos de aproximación en tiempo polinomial para problemas NP-completos, y la quinta sección presenta un esquema de aproximación en tiempo polinomial completo. La sección 35.1 comienza con un estudio del problema de cobertura de vértices, un problema de minimización NP-completo que tiene un algoritmo de aproximación con una relación de aproximación de 2. La sección 35.2 presenta un algoritmo de aproximación con una relación de aproximación de 2 para el caso del Problema del vendedor en el que la función de coste satisface el triángulo en igualdad. También muestra que sin la desigualdad del triángulo, para cualquier constante 1, no puede existir un algoritmo de aproximación a menos que P = NP. En la Sección 35.3, mostramos cómo usar un método voraz como un algoritmo de aproximación efectivo para el problema de cobertura de conjuntos, obteniendo una cobertura cuyo costo es, en el peor de los casos, un factor logarítmico mayor que el costo óptimo. La sección 35.4 presenta dos algoritmos de aproximación más. Primero estudiamos la versión de optimización de la satisfacibilidad de 3-CNF y proporcionamos un algoritmo aleatorio simple que produce una solución con una relación de aproximación esperada de  $8=7$ . Luego examinamos una variante ponderada del problema de cobertura de vértices y mostramos cómo usar la programación lineal para desarrollar un algoritmo de 2 aproximaciones. Finalmente, la Sección 35.5 presenta un esquema de aproximación en tiempo completamente polinomial para el problema de suma de subconjuntos.

---

### 35.1 El problema de la cobertura de vértices

La Sección 34.5.2 definió el problema de cobertura de vértices y lo demostró como NP-completo. Recuerde que una cubierta de vértice de un grafo no dirigido  $G$  . $V; E/$  es un subconjunto  $V'$  tal que si  $u; /$  es un  $^0$  borde de  $G$ , entonces  $u \in V'$  (o ambos). El tamaño de un  $^0$  o 2 voltios  $^0$  la cobertura de vértices es el número de vértices que tiene.

El problema de la cobertura de vértices consiste en encontrar una cobertura de vértices de tamaño mínimo en un grafo no dirigido dado. A tal cobertura de vértices la llamamos cobertura de vértices óptima. Este problema es la versión de optimización de un problema de decisión NP-completo.

Aunque no sabemos cómo encontrar una cobertura de vértices óptima en un gráfico  $G$  en tiempo polinomial, podemos encontrar eficientemente una cobertura de vértices que sea casi óptima.

El siguiente algoritmo de aproximación toma como entrada un gráfico no dirigido  $G$  y devuelve una cobertura de vértices cuyo tamaño se garantiza que no será más del doble del tamaño de una cobertura de vértices óptima.

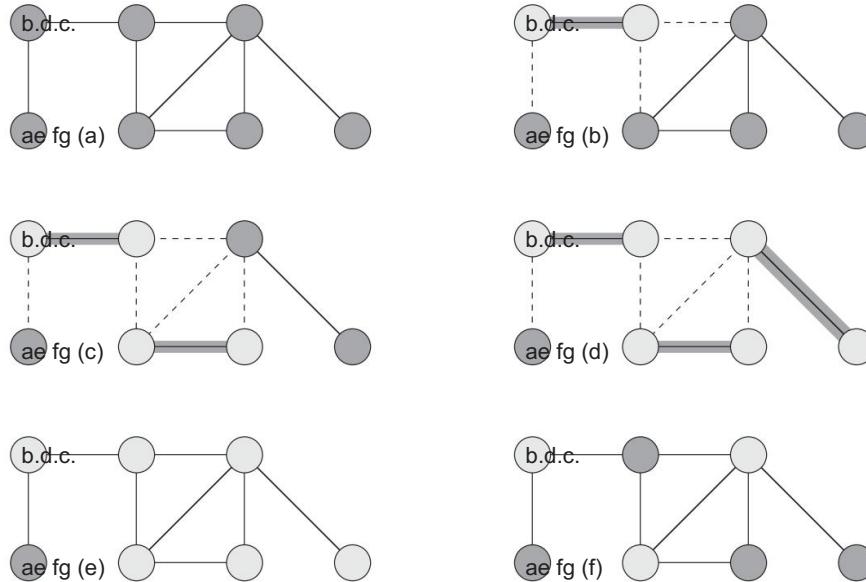


Figura 35.1 El funcionamiento de APPROX-VERTEX-COVER. (a) El gráfico de entrada  $G$ , que tiene 7 vértices y 8 aristas. (b) El borde  $b, c$ , que se muestra pesado, es el primer borde elegido por APPROX-VERTEX COVER. Los vértices  $b$  y  $c$ , que se muestran ligeramente sombreados, se agregan al conjunto  $C$  que contiene la cubierta de vértice que se está creando. Bordes  $a, e, f, g$ ; antes de Cristo;  $e, f$ ; y  $c, d$ ; que se muestran con guiones, se eliminan porque ahora están cubiertos por algún vértice en  $C$ . (c) Arista  $e, f$ ; se elige  $f$ ; los vértices  $e, f$  se suman a  $C$ . (d) Arista  $f, d$ ; se elige  $f$ ; los vértices  $d, g$  se suman a  $C$ . (e) El conjunto  $C$ , que es la cubierta de vértices producida por APPROX-VERTEX-COVER, contiene los seis vértices  $b, c, d, e, f, g$ . (f) La cobertura de vértices óptima para este problema contiene solo tres vértices:  $b, d$  y  $e$ .

### APPROX-VERTEX-COVER.G/

```

1 disco
compacto; 2 E0
DG:E 3 mientras
    que E0 ≠ ; dejarte; / sea una arista
    arbitraria de E0 4 5 CDC
    [ fu; g 6 eliminar de E0 cada borde que incide en u o 7 devolver C

```

La figura 35.1 ilustra cómo funciona APPROX-VERTEX-COVER en un gráfico de ejemplo. La variable  $C$  contiene la cubierta de vértice que se está construyendo. La línea 1 inicializa  $C$  en el conjunto vacío. La línea 2 establece que  $E0$  sea una copia del conjunto de aristas  $G:E$  del gráfico. El ciclo de las líneas 3 a 6 toma repetidamente un borde  $.u; / de E0$  agrega su

puntos finales  $u$  y  $C$ , y elimina todos los bordes en  $E_0$  que están cubiertos por  $u$  o  $C$ . Finalmente, la línea 7 devuelve la cobertura del vértice  $C$ . El tiempo de ejecución de este algoritmo es  $O(VE)$ , usando listas de adyacencia para representar  $E_0$ .

### Teorema 35.1

APPROX-VERTEX-COVER es un algoritmo de 2 aproximaciones en tiempo polinomial.

Prueba Ya hemos demostrado que APPROX-VERTEX-COVER se ejecuta en tiempo polinómico.

El conjunto  $C$  de vértices que devuelve APPROX-VERTEX-COVER es una cobertura de vértices, ya que el algoritmo realiza un bucle hasta que cada arista en  $G:E$  ha sido cubierta por algún vértice en  $C$ .

Para ver que APPROX-VERTEX-COVER devuelve una cobertura de vértice que es como mucho el doble del tamaño de una cubierta óptima, sea  $A$  el conjunto de aristas que eligió la línea 4 de APPROX-VERTEX-COVER. Para cubrir las aristas en  $A$ , cualquier cobertura de vértice, en particular, una cobertura óptima  $C$ , debe incluir al menos un extremo de cada arista en  $A$ . No hay dos aristas en  $A$  que comparten un extremo, ya que una vez que se selecciona una arista en línea 4, todas las demás aristas que inciden en sus extremos se eliminan de  $E_0$  en la línea 6. Por lo tanto, no hay dos aristas en  $A$  que estén cubiertas por el mismo vértice de  $C$ , y tenemos la cota inferior

$$|C| \geq |A| : \quad (35.2)$$

en el tamaño de una cubierta de vértice óptima. Cada ejecución de la línea 4 elige un borde para el cual ninguno de sus puntos finales ya está en  $C$ , lo que produce un límite superior (un límite superior exacto, de hecho) en el tamaño de la cubierta de vértice devuelta:

$$|C| \leq 2|A| : \quad (35.3)$$

Combinando las ecuaciones (35.2) y (35.3), obtenemos

$$|C| \leq 2|A| \leq 2|C| ;$$

demonstrando así el teorema. ■

Reflexionemos sobre esta prueba. Al principio, puede preguntarse cómo podemos probar que el tamaño de la cobertura de vértices devuelta por APPROX-VERTEX-COVER es como mucho el doble del tamaño de una cobertura de vértices óptima, cuando ni siquiera conocemos el tamaño de una cobertura de vértices óptima. En lugar de exigir que conozcamos el tamaño exacto de una cobertura de vértice óptima, confiamos en un límite inferior del tamaño. Como el ejercicio 35.1-2 le pide que muestre, el conjunto  $A$  de aristas que selecciona la línea 4 de APPROX-VERTEX-CUBIERTA es en realidad una coincidencia máxima en el gráfico  $G$ . (Una coincidencia máxima es una coincidencia que no es un subconjunto propio de ningún otra coincidencia.) El tamaño de una coincidencia máxima

es, como argumentamos en la demostración del Teorema 35.1, un límite inferior del tamaño de una cobertura de vértice óptima. El algoritmo devuelve una cobertura de vértice cuyo tamaño es como máximo el doble del tamaño de la coincidencia máxima A. Al relacionar el tamaño de la solución devuelta con el límite inferior, obtenemos nuestra relación de aproximación. También utilizaremos esta metodología en secciones posteriores.

### Ejercicios

#### 35.1-1

Dé un ejemplo de un gráfico para el cual APPROX-VERTEX-COVER siempre produce una solución subóptima.

#### 35.1-2

Demuestre que el conjunto de aristas seleccionadas en la línea 4 de APPROX-VERTEX-COVER forma una coincidencia máxima en el gráfico G.

#### 35.1-3 ?

El profesor Bündchen propone la siguiente heurística para resolver el problema de la cobertura de vértices. Seleccione repetidamente un vértice de mayor grado y elimine todos sus bordes incidentes. Dé un ejemplo para mostrar que la heurística del profesor no tiene una proporción de aproximación de 2. (Sugerencia: intente con un gráfico bipartito con vértices de grado uniforme a la izquierda y vértices de grado variable a la derecha).

#### 35.1-4

Proporcione un algoritmo codicioso eficiente que encuentre una cobertura de vértice óptima para un árbol en tiempo lineal.

#### 35.1-5

De la demostración del teorema 34.12, sabemos que el problema de cobertura de vértices y el problema de camarilla NP-completo son complementarios en el sentido de que una cobertura de vértice óptima es el complemento de una camarilla de tamaño máximo en el gráfico de complemento. ¿Esta relación implica que existe un algoritmo de aproximación en tiempo polinomial con una relación de aproximación constante para el problema de la camarilla? Justifica tu respuesta.

## 35.2 El problema del viajante de comercio

En el problema del viajante de comercio presentado en la Sección 34.5.4, tenemos un grafo no dirigido completo  $G = \langle V, E \rangle$  que tiene un costo entero no negativo  $c_{uv}$  asociado a cada arista  $(u, v) \in E$ , y debemos encontrar un ciclo hamiltoniano (un recorrido) de  $G$  con costo mínimo. Como una extensión de nuestra notación, sea  $c_A$  el costo total de las aristas en el subconjunto  $A \subseteq E$ :

cA/DX                    cu; / :  
                           .u;/2A

En muchas situaciones prácticas, la forma menos costosa de ir de un lugar  $u$  a un lugar  $w$  es ir directamente, sin pasos intermedios. Dicho de otra manera, eliminar una parada intermedia nunca aumenta el costo. Formalizamos esta noción diciendo que la función de costo  $c$  satisface la desigualdad del triángulo si, para todos los vértices  $u, v, w \in V$ ,

$c_{uv} \leq c_{uw} + c_{wv}$

La desigualdad del triángulo parece como si se mantuviera naturalmente y se satisface automáticamente en varias aplicaciones. Por ejemplo, si los vértices del gráfico son puntos en el plano y el costo de viajar entre dos vértices es la distancia euclídea ordinaria entre ellos, entonces se cumple la desigualdad del triángulo.

Además, muchas funciones de costo distintas de la distancia euclídea satisfacen la desigualdad del triángulo.

Como muestra el ejercicio 35.2-2, el problema del viajante de comercio es NP-completo incluso si queremos que la función de costo satisfaga la desigualdad triangular. Por lo tanto, no deberíamos esperar encontrar un algoritmo de tiempo polinomial para resolver este problema exactamente. En cambio, buscamos buenos algoritmos de aproximación.

En la Sección 35.2.1, examinamos un algoritmo de 2 aproximaciones para el problema del viajante de comercio con la desigualdad triangular. En la Sección 35.2.2, mostramos que sin la desigualdad triangular, no existe un algoritmo de aproximación en tiempo polinomial con una relación de aproximación constante a menos que sea PD NP.

### 35.2.1 El problema del viajante de comercio con la desigualdad triangular

Aplicando la metodología de la sección anterior, primero calcularemos una estructura, un árbol de expansión mínimo, cuyo peso da un límite inferior en la duración de un recorrido óptimo de un vendedor ambulante. Luego usaremos el árbol de expansión mínimo para crear un recorrido cuyo costo no sea más del doble del peso del árbol de expansión mínimo, siempre que la función de costo satisfaga la desigualdad del triángulo. El siguiente algoritmo implementa este enfoque, llamando al algoritmo de árbol de expansión mínima MST-PRIM de la Sección 23.2 como una subrutina. El parámetro  $G$  es un gráfico no dirigido completo y la función de costo  $c$  satisface la desigualdad triangular.

```

APROX-TSP-TOUR.G; c/1
seleccione un vértice r ∈ G:V para que sea un vértice
"raíz" 2 calcule un árbol generador mínimo T para G desde la
raíz r usando MST-PRIM.G; C; r/
3 sea H una lista de vértices, ordenados según la primera vez que se visitan
en un preorder tree walk de T
4 regresa el ciclo hamiltoniano H

```

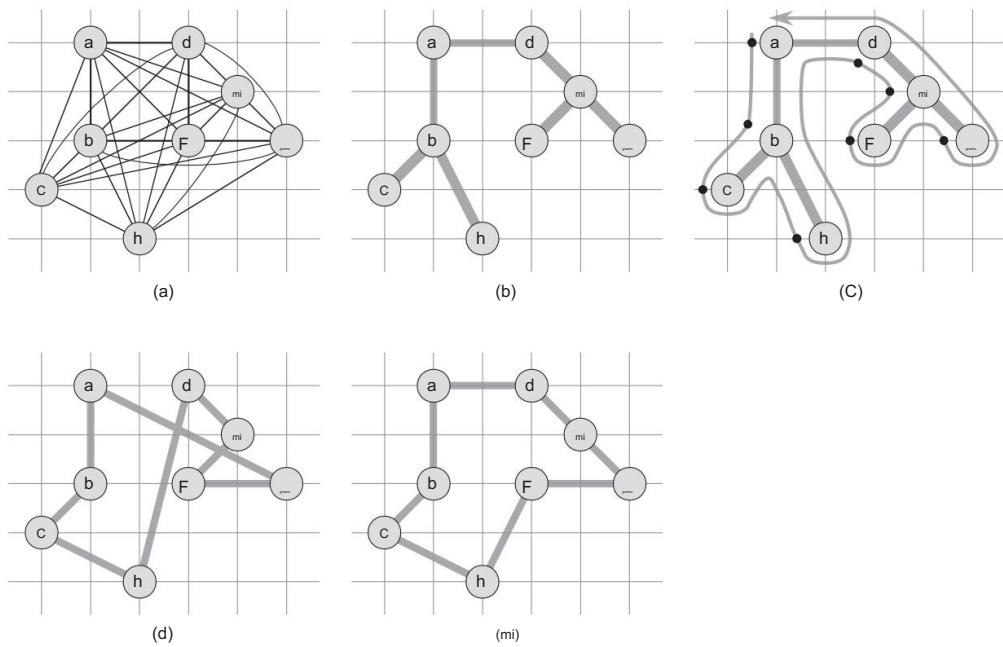


Figura 35.2 El funcionamiento de APPROX-TSP-TOUR. (a) Un grafo no dirigido completo. Los vértices se encuentran en intersecciones de líneas de cuadricula enteras. Por ejemplo, f está una unidad a la derecha y dos unidades arriba de h. La función de costo entre dos puntos es la distancia euclidiana ordinaria. (b) Un árbol de expansión mínimo T del gráfico completo, calculado por MST-PRIM. El vértice a es el vértice raíz. Solo se muestran los bordes en el árbol de expansión mínimo. Los vértices están etiquetados de tal manera que MST-PRIM los agrega al árbol principal en orden alfabetico. (c) Un paseo de T , comenzando en a. Un recorrido completo del árbol visita los vértices en el orden a; b; C; b; h; b; a; d; mi; F; mi; gramo; mi; d; a. Una caminata de preorder de T enumera un vértice justo cuando se encuentra por primera vez, como lo indica el punto al lado de cada vértice, lo que produce el orden a; b; C; h; d; mi; F; gramo. (d) Un recorrido obtenido al visitar los vértices en el orden dado por el recorrido de preorder, que es el recorrido H devuelto por APPROX-TSP-TOUR. Su coste total es de aproximadamente 19:074. (e) Un recorrido óptimo H para el gráfico completo original. Su coste total es de aproximadamente 14:715.

Recuerde de la Sección 12.1 que un recorrido de árbol de preorder visita recursivamente cada vértice en el árbol, enumerando un vértice cuando se encuentra por primera vez, antes de visitar cualquiera de sus hijos.

La figura 35.2 ilustra el funcionamiento de APPROX-TSP-TOUR. La parte (a) de la figura muestra un gráfico no dirigido completo, y la parte (b) muestra el árbol generador mínimo T crecido desde el vértice raíz a por MST-PRIM. La parte (c) muestra cómo un paseo preordenado de T visita los vértices, y la parte (d) muestra el recorrido correspondiente, que es el recorrido devuelto por APPROX-TSP-TOUR. La parte (e) muestra un recorrido óptimo, que es aproximadamente un 23% más corto.

Según el ejercicio 23.2-2, incluso con una implementación simple de MST-PRIM, el tiempo de ejecución de APPROX-TSP-TOUR es  $\sqrt{V^2}$ . Ahora mostramos que si la función de costo para una instancia del problema del viajante de comercio satisface la desigualdad del triángulo, entonces APPROX-TSP-TOUR devuelve un recorrido cuyo costo no es más del doble del costo de un recorrido óptimo.

### Teorema 35.2

APPROX-TSP-TOUR es un algoritmo de aproximación 2 en tiempo polinomial para el problema del vendedor ambulante con la desigualdad triangular.

**Prueba** Ya hemos visto que APPROX-TSP-TOUR se ejecuta en tiempo polinomial.

Sea  $H$  un recorrido óptimo para el conjunto de vértices dado. Obtenemos un árbol de expansión eliminando cualquier borde de un recorrido, y el costo de cada borde no es negativo. Por lo tanto, el peso del árbol de expansión mínimo  $T$  calculado en la línea 2 de APPROX-TSP TOUR proporciona un límite inferior del costo de un recorrido óptimo:  $c_T / c_H \geq 1$  : (35.4)

Una caminata completa de  $T$  enumera los vértices cuando se visitan por primera vez y también cada vez que se regresa después de una visita a un subárbol. Llamemos a este paseo completo  $W$ . El recorrido completo de nuestro ejemplo da la

orden a; b; C; b; h; b; a; d; mi; F; mi; gramo;

mi; d; a Dado que la caminata completa atraviesa cada arista de  $T$  exactamente dos veces, tenemos (extendiendo nuestra definición del costo  $c$  de manera natural para manejar múltiples conjuntos de aristas)  $c_W / c_T \leq 2$  : (35.5)

La desigualdad (35.4) y la ecuación (35.5) implican que

$$c_W / c_H \leq 2 \quad (35.6)$$

costo de  $W$  está dentro de un factor de 2 del costo de un recorrido óptimo.

Desafortunadamente, la caminata completa  $W$  generalmente no es un recorrido, ya que visita algunos vértices más de una vez. Sin embargo, por la desigualdad del triángulo, podemos eliminar una visita a cualquier vértice de  $W$  y el costo no aumenta. (Si eliminamos un vértice de  $W$  entre las visitas a  $u$  y  $w$ , el orden resultante especifica ir directamente de  $u$  a  $w$ ). Aplicando repetidamente esta operación, podemos eliminar de  $W$  todas las visitas a cada vértice excepto la primera. En nuestro ejemplo, esto deja el pedido a; b; C; h; d; mi; F; g : Este ordenamiento es el mismo que el obtenido

por un paseo en preorden del árbol  $T$ . Sea  $H$  el ciclo correspondiente a este paseo de preorden. Es un ciclo hamiltoniano, ya que cada

Cada vértice se visita exactamente una vez y, de hecho, es el ciclo calculado por APPROX TSP-TOUR. Dado que  $H$  se obtiene eliminando vértices del recorrido completo,  $W$  tiene , nosotros

$$cH/cW: \quad (35.7)$$

Combinando las desigualdades (35.6) y (35.7) se obtiene  $cH / 2c.H/$ , que completa la prueba. ■

A pesar de la buena relación de aproximación proporcionada por el Teorema 35.2, APPROX TSP-TOUR no suele ser la mejor opción práctica para este problema. Hay otros algoritmos de aproximación que suelen funcionar mucho mejor en la práctica. (Consulte las referencias al final de este capítulo).

### 35.2.2 El problema general del viajante de comercio

Si descartamos la suposición de que la función de costo  $c$  satisface la desigualdad triangular, entonces no podemos encontrar buenos recorridos aproximados en tiempo polinomial a menos que PD NP.

**Teorema 35.3 Si**

$P \neq NP$ , entonces para cualquier constante  $1$ , no existe un algoritmo de aproximación en tiempo polinomial con relación de aproximación para el problema general del viajante de comercio.

**Prueba** La prueba es por contradicción. Supongamos, por el contrario, que para algún número  $1$ , existe un algoritmo  $A$  de aproximación en tiempo polinomial con una proporción aproximada de berimación. Sin pérdida de generalidad, asumimos que es un número entero, redondeándolo si es necesario. Luego mostraremos cómo usar  $A$  para resolver instancias del problema del ciclo hamiltoniano (definido en la Sección 34.2) en tiempo polinomial.

Dado que el Teorema 34.13 nos dice que el problema del ciclo hamiltoniano es NP-completo, el Teorema 34.4 implica que si podemos resolverlo en tiempo polinomial, entonces PD NP.

Sea  $G(V; E)$  sea un ejemplo del problema del ciclo hamiltoniano. Deseamos determinar de manera eficiente si  $G$  contiene un ciclo hamiltoniano haciendo uso del algoritmo de aproximación hipotético  $A$ . Convertimos  $G$  en una instancia del problema del viajante de comercio de la siguiente manera. Sea  $G_0(V; E_0)$  sea el grafo completo sobre  $V$ ; eso es,

$E_0 = \{e_{uv} : u, v \in V\}$

asigne un costo entero a cada arista en  $E_0$  de la siguiente manera:

$$c_{uv} = \begin{cases} 1 & \text{si } (u, v) \in E \\ 2 & \text{si } (u, v) \notin E \end{cases}$$

Podemos crear representaciones de  $G_0$  y  $c$  a partir de una representación de  $G$  en un polinomio de tiempo en  $|V|$  y  $|E|$ .

Ahora, considere el problema del viajante de comercio .G0 ;c/. Si el gráfico original G tiene un ciclo hamiltoniano H, entonces la función de costo c asigna a cada arista de H un costo de 1, y así .G0 ;c/ contiene un recorrido de costo  $jV j$ . Por otro lado, si G no contiene un ciclo hamiltoniano, entonces cualquier vuelta a G0 debe usar alguna arista que no esté en E.

Pero cualquier recorrido que use una arista que no esté en E tiene un costo de al menos

$$. jV j C 1 / C .jV j 1 / D jV j C jV j > jV j$$

Debido a que las aristas que no están en G son tan costosas, existe una brecha de al menos  $jV j$  entre el costo de un recorrido que es un ciclo hamiltoniano en G (costo  $jV j$ ) y el costo de cualquier otro recorrido (costo de al menos  $jV j C jV j$ ). Por lo tanto, el costo de un recorrido que no es un ciclo hamiltoniano en G es al menos un factor de C 1 mayor que el costo de un recorrido que es un ciclo hamiltoniano en G.

Ahora, supongamos que aplicamos el algoritmo de aproximación A al problema del viajante de comercio .G0 ;c/. Debido a que se garantiza que A devolverá un recorrido de costo no mayor que el costo de un recorrido óptimo, si G contiene un ciclo hamiltoniano, entonces A debe devolverlo. Si G no tiene ciclo hamiltoniano, entonces A devuelve un recorrido de costo mayor que  $jV j$ . Por lo tanto, podemos usar A para resolver el problema del ciclo hamiltoniano en tiempo polinomial. ■

La demostración del teorema 35.3 sirve como ejemplo de una técnica general para demostrar que no podemos aproximarlos muy bien a un problema. Suponga que dado un problema NP-difícil X, podemos producir en tiempo polinomial un problema de minimización Y tal que las instancias "sí" de X corresponden a instancias de Y con valor como máximo k (para algún k), pero que "no" instancias de X corresponden a instancias de Y con valor mayor que k. Entonces, hemos demostrado que, a menos que P = NP, no existe un algoritmo de aproximación en tiempo polinomial para el problema Y

### Ejercicios

#### 35.2-1

Suponga que un grafo no dirigido completo GD .V; E/ con al menos 3 vértices tiene una función de costo c que satisface la desigualdad del triángulo. Demostrar que  $c(u) \leq 0$  para todos ustedes; 2 voltios

#### 35.2-2

Muestre cómo en tiempo polinomial podemos transformar una instancia del problema del viajante de comercio en otra instancia cuya función de costo satisfaga la desigualdad triangular. Las dos instancias deben tener el mismo conjunto de recorridos óptimos. Explique por qué tal transformación de tiempo polinomial no contradice el teorema 35.3, suponiendo que P ≠ NP.

## 35.2-3

Considere la siguiente heurística del punto más cercano para construir un recorrido aproximado de un vendedor ambulante cuya función de costo satisface la desigualdad triangular. Comience con un ciclo trivial que consta de un solo vértice elegido arbitrariamente. En cada paso, identifique el vértice  $u$  que no está en el ciclo pero cuya distancia a cualquier vértice del ciclo es mínima. Supongamos que el vértice del ciclo más cercano a  $u$  es el vértice  $v$ .

Amplíe el ciclo para incluir  $u$  insertando  $u$  justo después de  $v$ . Repita hasta que todos los vértices estén en el ciclo. Demuestre que esta heurística devuelve un recorrido cuyo costo total no es más del doble del costo de un recorrido óptimo.

## 35.2-4

En el problema del viajante con cuello de botella, deseamos encontrar el ciclo hamiltoniano que minimice el costo de la parte más costosa del ciclo. Suponiendo que la función de costo satisface la desigualdad triangular, demuestre que existe un algoritmo de aproximación de tiempo polinomial con relación de aproximación 3 para este problema. (Sugerencia: demuestre recursivamente que podemos visitar todos los nodos en un árbol de expansión de cuello de botella, como se discutió en el problema 23-3, exactamente una vez dando un paseo completo por el árbol y saltando nodos de ping, pero sin saltar más de dos nodos intermedios consecutivos).

Muestre que la arista más costosa en un árbol de expansión de cuello de botella tiene un costo que es, como mucho, el costo de la arista más costosa en un ciclo hamiltoniano de cuello de botella).

## 35.2-5

Suponga que los vértices de una instancia del problema del viajante de comercio son puntos en el plano y que el costo es  $c(u, v)$  es la distancia euclídea entre los puntos  $u$  y  $v$ . Demostrar que un recorrido óptimo nunca se cruza a sí mismo.

---

35.3 El problema de la cobertura del conjunto

El problema de cobertura de conjuntos es un problema de optimización que modela muchos problemas que requieren la asignación de recursos. Su problema de decisión correspondiente generaliza el problema de cobertura de vértices NP-completo y, por lo tanto, también es NP-difícil. Sin embargo, el algoritmo de aproximación desarrollado para manejar el problema de la cobertura de vértices no se aplica aquí, por lo que debemos probar otros enfoques. Examinaremos una heurística codiciosa simple con una relación de aproximación logarítmica. Es decir, a medida que aumenta el tamaño de la instancia, el tamaño de la solución aproximada puede crecer, en relación con el tamaño de una solución óptima. Sin embargo, debido a que la función logarítmica crece con bastante lentitud, este algoritmo de aproximación puede dar resultados útiles.

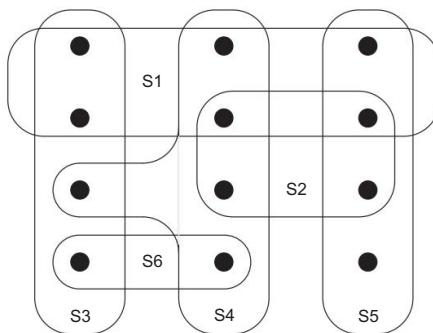


Figura 35.3 Una instancia  $.X; F/$  del problema de cobertura de conjuntos, donde  $X$  consta de los 12 puntos negros y  $F = \{S_1, S_2, S_3, S_4, S_5, S_6\}$ . Una cubierta de conjunto de tamaño mínimo es  $\{S_3, S_4, S_5, S_6\}$ , con tamaño 3. El algoritmo voraz produce una cubierta de tamaño 4 seleccionando los conjuntos  $S_1, S_4, S_5$  y  $S_3$  o los conjuntos  $S_1, S_4, S_5$  y  $S_6$ , en orden.

Una instancia  $.X; F/$  del problema de cobertura de conjuntos consta de un conjunto finito  $X$  y una familia  $F$  de subconjuntos de  $X$ , tal que cada elemento de  $X$  pertenece al menos a un subconjunto de  $F$ :

$$X \subseteq \bigcup_{S \in F} S$$

Decimos que un subconjunto  $S \subseteq F$  cubre sus elementos. El problema es encontrar un subconjunto  $C \subseteq F$  de tamaño mínimo cuyos miembros cubran todo  $X$ :

$$X \subseteq \bigcup_{S \in C} S \quad (35.8)$$

Decimos que cualquier  $C$  que satisface la ecuación (35.8) cubre  $X$ . La figura 35.3 ilustra el problema de la cobertura de conjuntos. El tamaño de  $C$  es el número de conjuntos que contiene, en lugar del número de elementos individuales en estos conjuntos, ya que cada subconjunto  $C$  que cubre  $X$  debe contener todos los elementos individuales  $x \in X$ . En la Figura 35.3, la cubierta mínima establecida tiene tamaño 3.

El problema de la cobertura de conjuntos abstrae muchos problemas combinatorios que surgen comúnmente. Como ejemplo simple, suponga que  $X$  representa un conjunto de habilidades que se necesitan para resolver un problema y que tenemos un conjunto dado de personas disponibles para trabajar en el problema. Deseamos formar un comité, que contenga la menor cantidad de personas posible, de modo que por cada habilidad requerida en  $X$ , al menos un miembro del comité tenga esa habilidad. En la versión de decisión del problema de cobertura de conjuntos, preguntamos si existe una cobertura con un tamaño como máximo  $k$ , donde  $k$  es un parámetro adicional especificado en la instancia del problema. La versión de decisión del problema es NP-completo, como se le pide que muestre en el ejercicio 35.3-2.

### Un algoritmo de aproximación codicioso

El método codicioso funciona seleccionando, en cada etapa, el conjunto  $S$  que cubre la mayor cantidad de elementos restantes que se descubren.

```

GREEDY-SET-COVER.X;F /1 UDX
2 CD ; 3
mientras U
  ; 4 seleccione un
  S 2 F que maximice  $\sum_{j \in U} f_j$ 
  UDUS 6 CDC [ fSg 7 devuelva
  C

```

En el ejemplo de la figura 35.3, GREEDY-SET-COVER suma a  $C$ , en orden, los conjuntos  $S_1, S_4$  y  $S_5$ , seguidos por  $S_3$  o  $S_6$ .

El algoritmo funciona de la siguiente manera. El conjunto  $U$  contiene, en cada etapa, el conjunto de elementos descubiertos restantes. El conjunto  $C$  contiene la cubierta que se está construyendo. La línea 4 es el paso codicioso de toma de decisiones, eligiendo un subconjunto  $S$  que cubra tantos elementos descubiertos como sea posible (rompiendo lazos arbitrariamente). Después de seleccionar  $S$ , la línea 5 elimina sus elementos de  $U$  y la línea 6 coloca  $S$  en  $C$ . Cuando termina el algoritmo, el conjunto  $C$  contiene una subfamilia de  $F$  que cubre  $X$ .

Podemos implementar fácilmente GREEDY-SET-COVER para ejecutar el polinomio de tiempo en  $|X|$  y  $|F|$ . Dado que el número de iteraciones del ciclo en las líneas 3 a 6 está acotado desde arriba por  $\min_j |X_j| / |F_j|$ , y podemos implementar el cuerpo del ciclo para que se ejecute en el tiempo  $O(|X| |F|)$ , una implementación simple se ejecuta en el tiempo  $O(|X| |F| \min_j |X_j| / |F_j|)$ . El ejercicio 35.3 pide un algoritmo de tiempo lineal.

### Análisis

Ahora mostramos que el algoritmo codicioso devuelve una cobertura de conjunto que no es mucho más grande que una cobertura de conjunto óptima. Por conveniencia, en este capítulo denotamos el número armónico  $H_d$   $H_d = \frac{1}{d}$  (ver Sección A.1) por  $H_d$  como un límite  $P_d$  condición, definimos  $H_0 = 0$ .

### El teorema 35.4

GREEDY-SET-COVER es un algoritmo de aproximación  $n/\ln n$  en tiempo polinomial, donde

$n/\ln n \cdot H_d \cdot \max_j f_j / \sum_j f_j$

Prueba Ya hemos demostrado que GREEDY-SET-COVER se ejecuta en tiempo polinomial.

Para mostrar que GREEDY-SET-COVER es un algoritmo de aproximación .n/, asignamos un costo de 1 a cada conjunto seleccionado por el algoritmo, distribuimos este costo entre los elementos cubiertos por primera vez y luego usamos estos costos para derivar la relación deseada entre el tamaño de una cobertura de conjunto óptima C y el tamaño de la cobertura de conjunto C devuelta por el algoritmo. Sea Si el i-ésimo subconjunto seleccionado por GREEDY-SET-COVER; el algoritmo incurre en un costo de 1 cuando agrega Si a C. Distribuimos este costo de seleccionar Si de manera uniforme entre los elementos cubiertos por primera vez por Si . Sea cx el costo asignado al elemento x, por cada  $x \in X$ . A cada elemento se le asigna un costo una sola vez, cuando se cubre por primera vez. Si x está cubierto por primera vez por Si , entonces

$$\frac{1}{\sum_{j \in S_i} c_j} \leq c_x \quad \text{para } x \in X$$

paso del algoritmo asigna 1 unidad de costo, y así

$$\sum_{x \in X} c_x \leq \sum_{j \in S_i} c_j \quad (35.9)$$

Cada elemento  $x \in X$  está en al menos un conjunto en la cobertura óptima C, por lo que tenemos

$$\sum_{x \in X} c_x \geq \sum_{j \in S_i} c_j \quad (35.10)$$

Combinando la ecuación (35.9) y la desigualdad (35.10), tenemos que

$$\sum_{x \in X} c_x = \sum_{j \in S_i} c_j \quad (35.11)$$

El resto de la prueba se basa en la siguiente desigualdad clave, que demostraremos en breve. Para cualquier conjunto S perteneciente a la familia F ,

$$\sum_{x \in X} c_x \leq \sum_{j \in S_i} c_j \quad (35.12)$$

De las desigualdades (35.11) y (35.12), se sigue que

$$\begin{aligned} &\sum_{x \in X} c_x \leq \sum_{j \in S_i} c_j \\ &\sum_{x \in X} c_x \leq \max_{j \in S_i} c_j \end{aligned}$$

demonstrando así el teorema.

Todo lo que queda es probar la desigualdad (35.12). Considere cualquier conjunto  $S \subseteq F$  y cualquier

yo  $D = \{S_1, S_2, \dots, S_k\}$ , y sea

$u(D, S) = \sum_{j \in S} c_j$

de elementos en S que quedan descubiertos después de que el algoritmo ha seleccionado los conjuntos  $S_1, S_2, \dots, S_k$ . Definimos  $u_0(D, S)$  como el número de elementos

de  $S$ , todos inicialmente descubiertos. Sea  $k$  el menor índice tal que  $u_k \in D_0$ , de modo que cada elemento de  $S$  esté cubierto por al menos uno de los conjuntos  $S_1; S_2; \dots; S_k$  y algún elemento en  $S$  es descubierto por  $S_1 \cup S_2 \cup \dots \cup S_{k-1}$ . Entonces, los elementos  $u_{i+1} u_i$  y  $u_{i+1} u_i$  de  $S$  son cubiertos por primera vez por  $S_i$ , para  $i = 1; 2; \dots; k$ .

De este modo,

$$\frac{\underset{x \in S}{\underset{i=1}{\overset{k}{\text{X cx } D_x . u_{i+1} u_i}}}}{\underset{j \in S}{\underset{i=1}{\overset{k}{\text{jSi . S}_1 \cup S_2 \cup \dots \cup S_{i-1}/j}}}}$$

Observa eso

$$\frac{\underset{D}{\underset{i=1}{\text{jSi . S}_1 \cup S_2 \cup \dots \cup S_{i-1}/j}}}{\underset{D}{\underset{i=1}{\text{jS . S}_1 \cup S_2 \cup \dots \cup S_{i-1}/j}}};$$

porque la elección codiciosa de  $S_i$  garantiza que  $S$  no puede cubrir más elementos nuevos que  $S_i$  (de lo contrario, el algoritmo habría elegido  $S$  en lugar de  $S_i$ ). En consecuencia, obtenemos

$$\frac{\underset{x \in S}{\underset{i=1}{\text{X cx X . u}_{i+1} u_i}}}{\underset{ui_1}{\text{ui}_1}}$$

Ahora limitamos esta cantidad de la siguiente manera:

$$\begin{aligned} &\frac{\underset{x \in S}{\underset{i=1}{\text{X cx X . u}_{i+1} u_i}}}{\underset{ui_1}{\text{ui}_1}} \\ &\quad \frac{\underset{\substack{k \\ i \in D_1}}{\underset{\substack{tu \\ j \in D_1 \cup C_1}}{\text{DX . X}_{i+1} \underset{\substack{1 \\ ui_1}}{\text{Xi}_1}}}} \\ &\quad \frac{\underset{\substack{k \\ i \in D_1}}{\underset{\substack{tu \\ j \in D_1 \cup C_1}}{\text{X} \underset{\substack{1 \\ j}}{\text{Xi}_1}}}} \quad (\text{porque } j \in u_i) \\ &\quad \frac{\underset{\substack{k \\ i \in D_1}}{\underset{\substack{tu \\ j \in D_1}}{\text{DX} \underset{\substack{1 \\ j}}{\text{Xi}_1}}}} \quad \frac{\underset{\substack{k \\ j \in D_1}}{\underset{\substack{x_{ui} \\ j}}{\text{1}}}}{\underset{\substack{k \\ j \in D_1}}{\underset{\substack{x_{ui} \\ j}}{\text{1}}}} \\ &\quad \frac{\underset{i \in D_1}{\text{DX . H.u}_0 / H.u_i / H.u_i / /}}{\underset{i \in D_1}{\text{D H.u}_0 / H.u_k / D}} \quad (\text{porque la suma se eleva}) \\ &\quad \frac{\underset{i \in D_1}{\text{H.u}_0 / H.0 / D}}{\underset{i \in D_1}{\text{H.jS}_j / ;}} \quad (\text{porque H.0 / D 0}) \end{aligned}$$

lo que completa la demostración de la desigualdad (35.12). ■

## Corolario 35.5

GREEDY-SET-COVER es un algoritmo de aproximación  $\ln jXjC1/$  de tiempo polinomial.

Prueba Utilice la desigualdad (A.14) y el Teorema 35.4. ■

En algunas aplicaciones,  $\max f(jS_j) WS \geq 2 F_g$  es una pequeña constante, por lo que la solución devuelta por GREEDY-SET-COVER es, como mucho, una pequeña constante veces mayor que la óptima. Una de estas aplicaciones ocurre cuando esta heurística encuentra una cobertura de vértice aproximada para un gráfico cuyos vértices tienen un grado máximo de 3. En este caso, la solución encontrada por GREEDY-SET-COVER no es más que  $H.3/D \leq 6$  veces tan grande como solución óptima, una garantía de rendimiento ligeramente mejor que la de APPROX-VERTEX-COVER.

## Ejercicios

## 35.3-1

Considere cada una de las siguientes palabras como un conjunto de letras: farid; estrellarse; drenar; escuchó; perdido; nariz; rehuir; pizarra; trampa; enhebrar. Muestre qué cobertura de conjunto produce GREEDY-SET-COVER cuando desempatamos a favor de la palabra que aparece primero en el diccionario.

## 35.3-2

Muestre que la versión de decisión del problema de cobertura de conjuntos es NP-completo reduciéndola del problema de cobertura de vértices.

## 35.3-3

Muestre cómo implementar GREEDY-SET-COVER de tal manera que se ejecute en el tiempo  $O(P \cdot S^2 F \cdot jS_j)$ .

## 35.3-4

Muestre que la siguiente forma más débil del teorema 35.4 es trivialmente cierta:

$jC_j \geq jC_j \max f(jS_j) WS \geq 2 F_g$ :

## 35.3-5

GREEDY-SET-COVER puede devolver varias soluciones diferentes, dependiendo de cómo rompamos los empates en la línea 4. Proporcione un procedimiento BAD-SET-COVER-INSTANCE.n/ que devuelva una instancia de n elementos del conjunto- problema de cobertura para el cual, dependiendo de cómo rompamos los empates en la línea 4, GREEDY-SET-COVER puede devolver un número de soluciones diferentes que es exponencial en n.

## 35.4 Aleatorización y programación lineal

En esta sección, estudiamos dos técnicas útiles para diseñar algoritmos de aproximación: la aleatorización y la programación lineal. Daremos un algoritmo aleatorio simple para una versión de optimización de la satisfacibilidad de 3-CNF, y luego usaremos la programación lineal para ayudar a diseñar un algoritmo de aproximación para una versión ponderada del problema de cobertura de vértices. Esta sección solo toca la superficie de estas dos poderosas técnicas. Las notas de los capítulos dan referencias para un estudio más profundo de estas áreas.

Un algoritmo de aproximación aleatoria para la satisfacibilidad de MAX-3-CNF

Así como algunos algoritmos aleatorios calculan soluciones exactas, algunos algoritmos aleatorios calculan soluciones aproximadas. Decimos que un algoritmo aleatorio para un problema tiene una relación de aproximación de  $\frac{C}{n}$  si, para cualquier entrada de tamaño  $n$ , el costo esperado  $C$  de la solución producida por el algoritmo aleatorio está dentro de un factor de  $\frac{C}{n}$  del costo  $C$  de una solución óptima:

$$\text{máximo } \frac{C}{n} : \frac{C}{C} \quad .n/ : \quad (35.13)$$

Llamamos a un algoritmo aleatorio que logra una relación de aproximación de  $\frac{C}{n}$  un algoritmo de aproximación aleatorio  $\frac{C}{n}$ . En otras palabras, un algoritmo de aproximación aleatoria es como un algoritmo de aproximación determinista, excepto que la relación de aproximación es para un costo esperado.

Una instancia particular de 3-CNF satisfacible, como se define en la Sección 34.4, puede o no ser satisfacible. Para que sea satisfacible, debe existir una asignación de las variables de manera que cada cláusula se evalúe como 1. Si una instancia no es satisfacible, es posible que deseemos calcular qué tan "cerca" de satisfacible está, es decir, es posible que deseemos encuentre una asignación de las variables que satisfaga tantas cláusulas como sea posible. Al problema de maximización resultante lo llamamos satisfacibilidad MAX-3-CNF. La entrada para la satisfacibilidad de MAX-3-CNF es la misma que para la satisfacibilidad de 3-CNF, y el objetivo es devolver una asignación de las variables que maximiza el número de cláusulas evaluadas en 1. Ahora mostramos que establecer aleatoriamente cada variable en 1 con probabilidad  $1/2$  y hasta 0 con probabilidad  $1/2$  produce un algoritmo de aproximación aleatorio  $8/7$ . De acuerdo con la definición de satisfacibilidad 3-CNF de la Sección 34.4, requerimos que cada cláusula conste de exactamente tres literales distintos. Suponemos además que ninguna cláusula contiene tanto una variable como su negación. (El ejercicio 35.4-1 le pide que elimine esta última suposición).

**Teorema 35.6**

Dada una instancia de satisfacibilidad de MAX-3-CNF con  $n$  variables  $x_1; x_2; \dots; x_n$  y  $m$  cláusulas, el algoritmo aleatorio que establece de forma independiente cada variable en 1 con probabilidad  $1/2$  y en 0 con probabilidad  $1/2$  es un algoritmo aleatorio de aproximación  $8/7$ .

Prueba Supongamos que hemos establecido independientemente cada variable en 1 con probabilidad  $1/2$  y en 0 con probabilidad  $1/2$ . Para  $i \in D[1; 2; \dots; m]$ , definimos la variable aleatoria del indicador

$Y_i = 1$  si la cláusula  $i$  se cumple ;

de modo que  $Y_i = 1$  siempre que hayamos puesto al menos uno de los literales en la  $i$ -ésima cláusula a 1. Como ningún literal aparece más de una vez en la misma cláusula, y como hemos supuesto que ninguna variable y su negación aparecen en la misma cláusula, las configuraciones de los tres literales en cada cláusula son independientes. Una cláusula no se cumple solo si sus tres literales se establecen en 0, por lo que  $\Pr[Y_i = 1] = 1/2$ . Así, tenemos  $\Pr[Y_i = 1] = 1/2$ . Por el Lema 5.1, tenemos  $E[Y_i] = 1/2$ . Sea  $Y$  el número total de cláusulas satisfechas, de modo que  $Y = Y_1 + Y_2 + \dots + Y_m$ . Entonces nosotros tenemos

$$E[Y] = E[\sum_{i=1}^m Y_i] = \sum_{i=1}^m E[Y_i] = m/2$$

$$E[Y] = E[\sum_{i=1}^m Y_i] = \sum_{i=1}^m E[Y_i] = m/2 \quad (\text{por linealidad de la expectativa})$$

$$\begin{aligned} E[Y] &= E[\sum_{i=1}^m Y_i] \\ &= \sum_{i=1}^m E[Y_i] \\ &= \sum_{i=1}^m \Pr[Y_i = 1] \\ &= m/2 \end{aligned}$$

Claramente,  $m$  es un límite superior en el número de cláusulas satisfechas y, por lo tanto, la relación de aproximación es como máximo  $m = 0,7m = 8/7$ .

**Aproximación de la cobertura de vértices ponderada mediante programación lineal**

En el problema de cobertura de vértices de peso mínimo, tenemos un grafo no dirigido  $G = (V, E)$  en la que cada vértice  $v \in V$  tiene un peso positivo asociado  $w_v$ . Para cualquier cubierta de  $V$ , definimos el peso de la cubierta de vértice  $w(V) = \sum_{v \in V} w_v$ . El objetivo es encontrar una cubierta de vértice de peso mínimo.

No podemos aplicar el algoritmo usado para la cobertura de vértices no ponderada, ni podemos usar una solución aleatoria; ambos métodos pueden devolver soluciones que están lejos de ser óptimas. Sin embargo, calcularemos un límite inferior para el peso del peso mínimo

cobertura de vértices, mediante el uso de un programa lineal. Luego “redondearemos” esta solución y la usaremos para obtener una cobertura de vértice.

Supongamos que asociamos una variable  $x_i$  con cada vértice  $v_i$  requiere, y déjanos que  $x_i$  sea igual a 0 o 1 para cada  $v_i$ . Ponemos en la cubierta de vértices si y sólo si  $x_i = 1$ . Entonces, podemos escribir la restricción que para cualquier arista  $u; v_i$ , al menos uno de  $u$  y debe estar en la cubierta de vértice como  $x_u + x_{v_i} \geq 1$ . Esta vista da lugar al siguiente programa entero 0-1 para encontrar un peso mínimo cubierta de vértice:

$$\begin{aligned} & \text{minimizar } X \text{ con } x_i \\ & \quad \forall v_i \\ & \text{sujeto a} \end{aligned} \tag{35.14}$$

$$x_u + x_{v_i} \geq 1 \quad \text{por cada } u; v_i \in V \tag{35.15}$$

$$x_i \in \{0, 1\} \quad \text{para cada } v_i \in V \tag{35.16}$$

En el caso especial en el que todos los pesos  $w_i$  son iguales a 1, esta formulación es la versión de optimización del problema de cobertura de vértices NP-hard. Sin embargo, suponga que eliminamos la restricción de que  $x_i \in \{0, 1\}$  y lo reemplazamos por  $0 \leq x_i \leq 1$ . Obtenemos entonces el siguiente programa lineal, que se conoce como relajación de programación lineal:

$$\begin{aligned} & \text{minimizar } X \text{ con } x_i \\ & \quad \forall v_i \\ & \text{sujeto a} \end{aligned} \tag{35.17}$$

$$x_u + x_{v_i} \leq 1 \quad \text{por cada } u; v_i \in V \tag{35.18}$$

$$x_i \geq 0 \quad \text{para cada } v_i \in V \tag{35.19}$$

$$0 \leq x_i \leq 1 \quad \text{para cada } v_i \in V \tag{35.20}$$

Cualquier solución factible del programa entero 0-1 en las líneas (35.14)–(35.16) también es una solución factible del programa lineal en las líneas (35.17)–(35.20). Por lo tanto, el valor de una solución óptima para el programa lineal da un límite inferior en el valor de una solución óptima para el programa entero 0-1 y, por lo tanto, un límite inferior en el peso óptimo en el problema de cobertura de vértice de peso mínimo.

El siguiente procedimiento utiliza la solución de la relajación de programación lineal para construir una solución aproximada al problema de cobertura de vértices de peso mínimo:

APROX-MIN-PESO-VC.G; con 1 CD; 2  
 calcule  $x_N$ ,  
 una solución óptima para el programa lineal en las líneas (35.17)–(35.20) 3 para cada 2 V 4 si  
 $x_i/N \leq 2$  5 CDC [ fg  
 6 devuelva C

El procedimiento APPROX-MIN-WEIGHT-VC funciona de la siguiente manera. La línea 1 inicializa la cubierta del vértice para que esté vacía. La línea 2 formula el programa lineal en las líneas (35.17)–(35.20) y luego resuelve este programa lineal. Una solución óptima le da a cada vértice un valor asociado  $x_i/N$  donde  $0 \leq x_i \leq 1$ . Usamos este valor para guiar la elección de qué vértices agregar a la cubierta de vértice C en las líneas 3–5. Si  $x_i/N \leq 2$ , sumamos a C; de lo contrario no lo hacemos. En efecto, estamos “redondeando” cada variable fraccionaria en la solución del programa lineal a 0 o 1 para obtener una solución al problema entero 0-1 en las líneas (35.14)–(35.16). Finalmente, la línea 6 devuelve la cubierta de vértice C.

El algoritmo del

teorema 35.7 APPROX-MIN-WEIGHT-VC es un algoritmo de 2 aproximaciones en tiempo polinomial para el problema de cobertura de vértices de peso mínimo.

Prueba Debido a que hay un algoritmo de tiempo polinomial para resolver el programa lineal en la línea 2, y debido a que el ciclo for de las líneas 3 a 5 se ejecuta en tiempo polinomial, APPROX . PESO MÍN.-VC es un algoritmo de tiempo polinomial.

Ahora mostramos que APPROX-MIN-WEIGHT-VC es un algoritmo de 2 aproximaciones. Sea C una solución óptima al problema de cobertura de vértices de peso mínimo, y sea  $w_C$  el valor de una solución óptima al programa lineal en las líneas (35.17)–(35.20). Dado que una cubierta óptima de vértices es una solución factible del programa lineal,  $w_C$  debe ser un límite inferior en  $w_C'$ , es decir,

$$w_C' \leq w_C \quad (35.21)$$

A continuación, afirmamos que al redondear los valores fraccionarios de las variables  $x_i$ ,  $N$  se produce un conjunto C que es una cubierta de vértices y satisface  $w_C' \leq w_C$ . Para ver que C es una cubierta de vértice, considere cualquier arista  $e = (u, v)$ . Por la restricción (35.18), sabemos que  $x_u/N \geq 1/2$ , lo que implica que al menos uno de  $x_u/N$  y  $x_v/N$  es al menos  $1/2$ . Por lo tanto, al menos uno de  $u$  y  $v$  está incluido en la cubierta de vértices, por lo que todas las aristas están cubiertas.

Ahora, consideraremos el peso de la cubierta. Tenemos

' DX c./ x./ N  
2V

X w./ x./ norte  
2V W Nx./1=2

X c./ 2V 1  
W Nx./1=2 1 2

DX con/  
2C 2  
1  
D 2 X con/  
2C

D 2 baño / : (35.22)

Combinando las desigualdades (35.21) y (35.22) se obtiene

wC / 2' 2w.C/ ;

y por lo tanto APPROX-MIN-WEIGHT-VC es un algoritmo de 2 aproximaciones. ■

### Ejercicios

#### 35.4-1

Demuestre que incluso si permitimos que una cláusula contenga tanto una variable como su negación, establecer aleatoriamente cada variable en 1 con probabilidad 1=2 y en 0 con probabilidad 1=2 aún produce un algoritmo de aproximación aleatorio 8=7.

#### 35.4-2

El problema de satisfacibilidad de MAX-CNF es como el problema de satisfacibilidad de MAX-3-CNF, excepto que no restringe que cada cláusula tenga exactamente 3 literales. Proporcione un algoritmo aleatorio de 2 aproximaciones para el problema de satisfacibilidad MAX-CNF.

#### 35.4-3

En el problema MAX-CUT, tenemos un grafo no dirigido no ponderado GD .V; MI/. Definimos un corte .S; VS/ como en el Capítulo 23 y el peso de un corte como el número de aristas que cruzan el corte. El objetivo es encontrar un corte de peso máximo. Supongamos que para cada vértice colocamos aleatoria e independientemente en S con probabilidad 1=2 y en VS con probabilidad 1=2. Demuestre que este algoritmo es un algoritmo aleatorio de 2 aproximaciones.

## 35.4-4

Muestre que las restricciones en la línea (35.19) son redundantes en el sentido de que si las eliminamos del programa lineal en las líneas (35.17)–(35.20), cualquier solución óptima para el programa lineal resultante debe satisfacer  $x_i \geq 0$  para cada  $i$ .

---

## 35.5 El problema de la suma de subconjuntos

Recuerde de la Sección 34.5.5 que una instancia del problema de suma de subconjuntos es un par  $(S, t)$ , donde  $S$  es un conjunto  $\{x_1, x_2, \dots, x_n\}$  de enteros positivos y  $t$  es un entero positivo. Este problema de decisión pregunta si existe un subconjunto de  $S$  que sume exactamente el valor objetivo  $t$ . Como vimos en la Sección 34.5.5, este problema es NP-completo.

El problema de optimización asociado con este problema de decisión surge en aplicaciones prácticas. En el problema de optimización, deseamos encontrar un subconjunto de  $\{x_1, x_2, \dots, x_n\}$  cuya suma sea lo más grande posible pero no mayor que  $t$ . Por ejemplo, podemos tener un camión que no puede transportar más de  $t$  libras y  $n$  cajas diferentes para enviar, la  $i$ -ésima de las cuales pesa  $x_i$  libras. Deseamos llenar el camión con la carga más pesada posible sin exceder el límite de peso dado.

En esta sección, presentamos un algoritmo de tiempo exponencial que calcula el valor óptimo para este problema de optimización y luego mostramos cómo modificar el algoritmo para que se convierta en un esquema de aproximación de tiempo polinomial completo. (Recuerde que un esquema de aproximación de tiempo completamente polinomial tiene un tiempo de ejecución que es polinomial en  $1/\epsilon$  así como en el tamaño de la entrada).

## Un algoritmo exacto en tiempo exponencial

Supongamos que calculamos, para cada subconjunto  $S_0$  de  $S$ , la suma de los elementos en  $S_0$  y luego seleccionamos, entre los subconjuntos cuya suma no excede de  $t$ , aquel cuya suma fue más cercana a  $t$ . Claramente, este algoritmo devolvería la solución óptima, pero podría llevar un tiempo exponencial. Para implementar este algoritmo, podríamos usar un procedimiento iterativo que, en la iteración  $i$ , calcule las sumas de todos los subconjuntos de  $\{x_1, x_2, \dots, x_i\}$ , utilizando como punto de partida las sumas de todos los subconjuntos de  $\{x_1, x_2, \dots, x_{i-1}\}$ . Al hacerlo, nos daríamos cuenta de que una vez que un subconjunto particular  $S_0$  tuviera una suma superior a  $t$ , no habría razón para mantenerlo, ya que ningún superconjunto de  $S_0$  podría ser la solución óptima. Ahora damos una implementación de esta estrategia.

El procedimiento EXACT-SUBSET-SUM toma un conjunto de entrada  $S$  y un valor objetivo  $t$ ; veremos su pseudocódigo en un momento. Este procedimiento es-

calcula de forma iterativa  $L_i$ , la lista de sumas de todos los subconjuntos de  $f x_1; \dots; x_i g$  que no superan  $t$ , y luego devuelve el valor máximo en  $L_n$ .

Si  $L$  es una lista de enteros positivos y  $x$  es otro entero positivo, entonces  $L[x]$  denota la lista de enteros derivados de  $L$  aumentando cada elemento de  $L$  en  $x$ . Por ejemplo, si  $L = h 1; 2; 3; 5; 9i$ , luego  $L[2] = D h 3; 4; 5; 7; 11i$ . También usamos esta notación para conjuntos, de modo que

$S C x D f s C x W s 2 Sg$ :

También usamos un procedimiento auxiliar  $\text{MERGE-LISTS}(L; L')$ , que devuelve la lista ordenada que es la fusión de sus dos listas de entrada ordenadas  $L$  y  $L'$  con los valores duplicados eliminados. Al igual que el procedimiento  $\text{MERGE}$  que usamos en la ordenación por fusión (Sección 2.3.1),  $\text{MERGE-LISTS}$  se ejecuta en el tiempo  $O(jLj C jL'j)$ . Omitimos el pseudocódigo para  $\text{MERGE LISTS}$ .

#### SUBCONJUNTO-EXACTA-

$SUM.S; t / 1 n$

$D jSj 2 L0 D h0i$

$3 \text{ para } i \text{ D } 1 \text{ to } n \text{ 4 } Li$

$D \text{ MERGE-LISTS}(Li; Li) C xi / 5$  elimina de  $Li$  todo elemento que sea mayor que  $t$  6 devuelve el elemento más grande en  $Ln$

Para ver cómo funciona  $\text{EXACT-SUBSET-SUM}$ , permita que  $P_i$  denote el conjunto de todos los valores obtenidos al seleccionar un subconjunto (posiblemente vacío) de  $f x_1; x_2; \dots; x_i g$  y sumando sus miembros. Por ejemplo, si  $S = f 1; 4; 5g$ , entonces

$P_1 = f 0; 1g$ ;

$P_2 = f 0; 1; 4; 5g$ ;

$P_3 = f 0; 1; 4; 5; 6; 9; 10g$ :

Dada la identidad

$$P_i = P_{i-1} \cup [ .P_{i-1} C x_i / ] \quad (35.23)$$

podemos demostrar por inducción sobre  $i$  (vea el ejercicio 35.5-1) que la lista  $L_i$  es una lista ordenada que contiene todos los elementos de  $P_i$  cuyo valor no es mayor que  $t$ . Dado que la longitud de  $L_i$  puede ser tanto como  $2^i$   $\text{EXACT-SUBSET-SUM}$  es un algoritmo de tiempo exponencial en general, aunque es un algoritmo de tiempo polinomial en los casos especiales en los que  $t$  es polinomial en  $jSj$  o todos los números en  $S$  están acotados por un polinomio en  $jSj$ .

Un esquema de aproximación de tiempo completamente polinomial

Podemos derivar un esquema de aproximación de tiempo polinomial completo para el problema de suma de subconjuntos "recortando" cada lista  $L_i$  después de crearla. La idea detrás del recorte es

que si dos valores en L están cerca uno del otro, entonces como solo queremos una solución aproximada, no necesitamos mantener ambos explícitamente. Más precisamente, usamos un parámetro de recorte  $i$  tal que  $0 < i < 1$ . Cuando recortamos una lista L por  $i$ , eliminamos tantos elementos de L como sea posible, de tal manera que si  $L_0$  es el resultado de recortar L, entonces por cada elemento  $y$  que se eliminó de L, todavía hay un elemento  $'$  en  $L_0$  que se aproxima a  $y$ , es decir,

$$\frac{y}{1 - C} \leq y' \quad (35.24)$$

Podemos pensar en tal  $'$  como "representando" y en la nueva lista  $L_0$ . Cada elemento eliminado y está representado por un elemento restante  $'$  que satisface la desigualdad (35.24). Por ejemplo, si  $i = 0.1$

$DL \ h10; 11; 12; 15; 20; 21; 22; 23; 24; 29;$

entonces podemos recortar L para obtener

$L_0 \ Dh10; 12; 15; 20; 23; 29;$

donde el valor eliminado 11 está representado por 10, los valores eliminados 21 y 22 están representados por 20 y el valor eliminado 24 está representado por 23. Debido a que cada elemento de la versión recortada de la lista también es un elemento de la versión original de la lista, el recorte puede disminuir drásticamente la cantidad de elementos guardados mientras se mantiene un valor representativo cercano (y un poco más pequeño) en la lista para cada elemento eliminado.

El siguiente procedimiento ajusta la lista LD  $hy1; y2; \dots; ym$  en el tiempo  $,.m/$ , dados L e  $i$ , y suponiendo que L se clasifica en un orden monótonamente creciente. El resultado del procedimiento es una lista recortada y ordenada.

TRIM.L;  $i / 1$

sea  $m$  la longitud de L 2 L0 D

$hy1i$  3 último D

$y1 4$  para  $i \ D 2$

a  $m 5$  si  $y1 >$  último .1

C  $i / / y1$  6 agregue  $y1$  al final de L0 último D      último porque L está ordenado

$y1 7 8$  retorno L0

El procedimiento escanea los elementos de L en orden monótonamente creciente. Un número se agrega a la lista devuelta L0 solo si es el primer elemento de L o si no puede ser representado por el número más reciente colocado en L0

Dado el procedimiento TRIM, podemos construir nuestro esquema de aproximación de la siguiente manera. Este procedimiento toma como entrada un conjunto SD  $fx1; x2; \dots; xng$  de n enteros (en orden arbitrario), un entero objetivo  $t$  y un "parámetro de aproximación" donde

0<<1: (35.25)

Devuelve un valor ' cuyo valor está dentro de un factor de 1 C de la solución óptima.

APROX-SUBSET-SUM.S; t; / 1 n

D jSj 2 L0 D

h0i 3 for i D 1

to n Li D MERGE-

LISTS.Li1; Li1 C xi/ 4 5 Li D TRIM.Li; =2n/

6 quita de Li todo elemento que

sea mayor que t 7 sea ' el valor más grande en Ln 8 devuelve '

La línea 2 inicializa la lista L0 para que sea la lista que contiene solo el elemento 0. El ciclo for en las líneas 3 a 6 calcula Li como una lista ordenada que contiene una versión adecuadamente recortada del conjunto Pi , con todos los elementos mayores que t eliminados. Dado que creamos Li a partir de Li1, debemos asegurarnos de que el recorte repetido no introduzca demasiada inexactitud compuesta. En un momento, veremos que APPROX-SUBSET-SUM devuelve una aproximación correcta, si existe.

Como ejemplo, supongamos que tenemos la

instancia SD h104; 102;

201; 101i con t D 308 y D 0:40. El parámetro de recorte i es =8 D 0:05. APPROX SUBSET-SUM calcula los siguientes valores en las líneas indicadas:

línea 2: L0 D h0i ;

línea 4: L1 D h0; 104i; línea

5: L1 D h0; 104i; línea 6: L1

D h0; 104i;

línea 4: L2 D h0; 102; 104; 206i; línea 5:

L2 D h0; 102; 206i; línea 6: L2 D

h0; 102; 206i;

línea 4: L3 D h0; 102; 201; 206; 303; 407i; línea 5:

L3 D h0; 102; 201; 303; 407i; línea 6: L3 D

h0; 102; 201; 303i;

línea 4: L4 D h0; 101; 102; 201; 203; 302; 303; 404i; línea 5:

L4 D h0; 101; 201; 302; 404i; línea 6: L4 D

h0; 101; 201; 302i;

El algoritmo devuelve ' D 302 como su respuesta, que está dentro del D 40 % de la respuesta óptima 307 D 104 C 102 C 101; de hecho, está dentro del 2%.

El teorema 35.8

APPROX-SUBSET-SUM es un esquema de aproximación en tiempo completamente polinomial para el problema de la suma de subconjuntos.

Prueba Las operaciones de recortar Li en la línea 5 y quitar de Li todo elemento que sea mayor que t mantienen la propiedad de que todo elemento de Li es también miembro de Pi. Por lo tanto, el valor ' devuelto en la línea 8 es de hecho la suma de algún subconjunto de S. Sea y 2 Pn una solución óptima al problema de la suma de subconjuntos. Entonces, de la línea 6, sabemos que ' y. Por la desigualdad (35.1), necesitamos demostrar que y= 1 C . También debemos mostrar que el tiempo de ejecución de este algoritmo es polinomial tanto en 1= como en el tamaño de la entrada.

Como se le pide que muestre en el Ejercicio 35.5-2, para cada elemento y en Pi que es como máximo t, existe un elemento ' 2 Li tal que

$$\overline{y \cdot 1 C} = 2n/i \quad ' y : \quad (35.26)$$

La desigualdad (35.26) debe cumplirse para y 2 Pn, y por lo tanto existe un elemento ' 2 Ln tal que

$$\overline{y \cdot 1 C} = 2n/n \quad ' y ;$$

y por lo tanto

$$\frac{y}{2} \quad 1 C \frac{n}{2n} \quad ' \quad (35.27)$$

Como existe un elemento ' 2 Ln que cumple la desigualdad (35.27), la desigualdad debe cumplirse para ', que es el valor más grande de Ln; eso es,

$$\frac{y}{2} \quad 1 C \frac{n}{2n} \quad ' \quad (35.28)$$

Ahora demostramos que y= 1 C . Lo hacemos demostrando que .1 C =2n/n 1 C . Por la ecuación (3.14), tenemos  $\lim_{n \rightarrow \infty} 1 C = 2n/n = e^2$ . El ejercicio 35.5-3 le pide que demuestre que

$$\frac{d}{dn} \quad 1 C \frac{n}{2n} \quad > 0: \quad (35.29)$$

Por tanto, la función .1 C =2n/n crece con n a medida que se acerca a su límite de e=2, y tenemos

$$\frac{1}{2n} C$$

$$e=2$$

$$1 \leq 2C \cdot e^2 / 2 \text{ (por desigualdad (3.13))}$$

$$1 \leq C \text{ (por la desigualdad (35.25)) .} \quad (35.30)$$

La combinación de las desigualdades (35.28) y (35.30) completa el análisis de la razón de aproximación.

Para mostrar que APPROX-SUBSET-SUM es un esquema de aproximación de tiempo completamente polinomial, derivamos un límite en la longitud de  $L_i$ . Después de recortar, los elementos sucesivos '0' y '0' de  $L_i$  deben tener la relación ' $0 = > 1C = 2n$ '. Es decir, deben diferir por un factor de al menos  $1C = 2n$ . Cada lista, por tanto, contiene el valor 0, posiblemente el valor 1, y hasta  $\log 1C = 2n$  t ~ valores adicionales. El número de elementos en cada lista  $L_i$  es como máximo

$$\begin{aligned} \log 1C = 2n &\leq t \leq 2D \\ &\leq \frac{\ln t}{\ln 2} C \\ &\leq \frac{2n}{\ln t} C \text{ (por desigualdad (3.17))} \\ &< \frac{3n \ln t}{\ln 2} C \text{ (por la desigualdad (35.25)) .} \end{aligned}$$

Este límite es polinomial en el tamaño de la entrada, que es el número de bits  $\lg t$  necesarios para representar  $t$  más el número de bits necesarios para representar el conjunto  $S$ , que a su vez es polinomial en  $n$  y en  $1=$ . Dado que el tiempo de ejecución de APPROX-SUBSET SUM es polinomial en las longitudes de  $L_i$ , concluimos que APPROX-SUBSET SUM es un esquema de aproximación de tiempo completamente polinomial. ■

### Ejercicios

#### 35.5-1

Demuestre la ecuación (35.23). Luego muestre que después de ejecutar la línea 5 de EXACT-SUBSET SUM,  $L_i$  es una lista ordenada que contiene todos los elementos de  $P_i$  cuyo valor no es mayor que  $t$ .

#### 35.5-2

Usando inducción en  $i$ , demuestre la desigualdad (35.26).

#### 35.5-3

Demostrar la desigualdad (35.29).

35.5-4

¿Cómo modificaría el esquema de aproximación presentado en esta sección para encontrar una buena aproximación al valor más pequeño no menor que  $t$  que sea la suma de algún subconjunto de la lista de entrada dada?

35.5-5

Modifique el procedimiento APPROX-SUBSET-SUM para que también devuelva el subconjunto de  $S$  que suma el valor  $t$ .

## Problemas

35-1 Empaquetado en

contenedores Suponga que se nos da un conjunto de  $n$  objetos, donde el tamaño si del  $i$ -ésimo objeto satisface  $0 < s_i < 1$ . Deseamos empacar todos los objetos en el número mínimo de contenedores de tamaño unitario. Cada contenedor puede contener cualquier subconjunto de los objetos cuyo tamaño total no exceda de 1.

- a. Demuestre que el problema de determinar el número mínimo de contenedores requeridos es NP-duro. (Sugerencia: reduzca del problema de suma de subconjuntos).

La heurística de primer ajuste toma cada objeto por turno y lo coloca en el primer contenedor que puede acomodarlo. Sea  $SD$  Pn iD1 si .

- b. Argumente que el número óptimo de contenedores requeridos es al menos  $dSe$ .

C. Argumente que la heurística del primer ajuste deja a lo sumo un recipiente menos de la mitad.

- d. Demuestre que el número de contenedores utilizados por la heurística de primer ajuste nunca es más que  $d2Se$ .

mi. Demuestre una relación de aproximación de 2 para la heurística de primer ajuste.

F. Dar una implementación eficiente de la heurística de primer ajuste y analizar su funcionamiento tiempo.

35-2 Aproximación al tamaño de una camarilla máxima Sea  $GD$  .V; E/

sea un grafo no dirigido. Para cualquier  $k \geq 1$ , defina  $G_k$  como el grafo no dirigido  $.V .k/; E_k/$ , donde  $V .k/$  es el conjunto de todas las  $k$ -tuplas ordenadas de vértices de  $V$  y  $E_k$  se define de modo que  $.v_1; v_2; \dots; v_k/$  es adyacente a  $.w_1; w_2; \dots; w_k/$  si y solo si para  $i \in \{1, 2, \dots, k\}$ , el vértice  $v_i$  es adyacente a  $w_i$  en  $G$ , o bien

$\sum_{i=1}^k D(v_i) \geq k$

- a. Demostrar que el tamaño de la camarilla máxima en  $G_k$  es igual a la  $k$ -ésima potencia del tamaño de la camarilla máxima en  $G$ .
- b. Argumente que si hay un algoritmo de aproximación que tiene una razón de aproximación constante para encontrar una camarilla de tamaño máximo, entonces hay un esquema de aproximación de tiempo polinomial para el problema.

### 35-3 Problema de cobertura de conjunto ponderado

Suponga que generalizamos el problema de cobertura de conjunto de manera que cada conjunto  $S_i$  en la familia  $F$  tiene un peso asociado  $w_i$  y el peso de una cubierta  $C$  es  $\sum_{S_i \in C} w_i$ .

Deseamos determinar una cubierta de peso mínimo. (La Sección 35.3 trata el caso en el que  $w_i = 1$  para todo  $i$ .)

Muestre cómo generalizar la heurística codiciosa de cobertura de conjuntos de una manera natural para proporcionar una solución aproximada para cualquier instancia del problema ponderado de cobertura de conjuntos. Demuestre que su heurística tiene una relación de aproximación de  $\frac{H_d}{d}$ , donde  $d$  es el tamaño máximo de cualquier conjunto  $S_i$ .

### 35-4 Coincidencia máxima Recuerde

que para un grafo no dirigido  $G$ , una coincidencia es un conjunto de aristas tal que dos aristas del conjunto no inciden en el mismo vértice. En la Sección 26.3, vimos cómo encontrar una coincidencia máxima en un gráfico bipartito. En este problema, veremos coincidencias en grafos no dirigidos en general (es decir, no se requiere que los grafos sean bipartitos).

- a. Una coincidencia máxima es una coincidencia que no es un subconjunto propio de ninguna otra coincidencia. Muestre que una coincidencia máxima no necesita ser una coincidencia máxima mostrando un gráfico no dirigido  $G$  y una coincidencia máxima  $M$  en  $G$  que no es una coincidencia máxima.  
(Sugerencia: puede encontrar un gráfico de este tipo con solo cuatro vértices).
- b. Considere un grafo no dirigido  $G$ . Dar un algoritmo codicioso OE-time ritmo para encontrar una coincidencia máxima en  $G$ .

En este problema, nos concentraremos en un algoritmo de aproximación de tiempo polinomial para una coincidencia máxima. Mientras que el algoritmo conocido más rápido para la coincidencia máxima toma un tiempo superlineal (pero polinomial), el algoritmo de aproximación aquí se ejecutará en un tiempo lineal. Mostrará que el algoritmo codicioso de tiempo lineal para la coincidencia máxima en la parte (b) es un algoritmo de aproximación 2 para la coincidencia máxima.

- C. Muestre que el tamaño de una coincidencia máxima en  $G$  es un límite inferior en el tamaño de cualquier cubierta de vértice para  $G$ .

d. Considere una coincidencia máxima M en  $G \setminus V(M)$ . Dejar

$T \setminus f \setminus V(W)$  alguna arista en M incide en g :

¿Qué puedes decir sobre el subgrafo de G inducido por los vértices de G que no están en T?

mi. Concluya de la parte (d) que  $2|M|$  es el tamaño de una cubierta de vértice para G.

F. Usando las partes (c) y (e), demuestre que el algoritmo codicioso en la parte (b) es un 2 aprox.

Algoritmo de imation para la máxima coincidencia.

### 35-5 Programación de máquinas paralelas En el

problema de programación de máquinas paralelas, tenemos n trabajos,  $J_1; J_2; \dots; J_n$ , donde cada trabajo  $J_k$  tiene asociado un tiempo de procesamiento no negativo de  $p_k$ . También tenemos m máquinas idénticas,  $M_1; M_2; \dots; M_m$ . Cualquier trabajo puede ejecutarse en cualquier máquina. Un programa especifica, para cada trabajo  $J_k$ , la máquina en la que se ejecuta y el período de tiempo durante el cual se ejecuta. Cada trabajo  $J_k$  debe ejecutarse en alguna máquina  $M_i$  durante  $p_k$  unidades de tiempo consecutivas, y durante ese período de tiempo ningún otro trabajo puede ejecutarse en  $M_i$ . Sea  $C_k$  el tiempo de finalización del trabajo  $J_k$ , es decir, el tiempo en el que el trabajo  $J_k$  completa su procesamiento. Dado un horario, definimos  $C_{\max} = \max_j C_j$  como el intervalo de tiempo del horario. El objetivo es encontrar un horario cuyo makepan sea mínimo.

Por ejemplo, supongamos que tenemos dos máquinas  $M_1$  y  $M_2$  y que tenemos cuatro trabajos  $J_1; J_2; J_3; J_4$ , con  $p_1 = 2, p_2 = 3, p_3 = 4$  y  $p_4 = 5$ . Luego se ejecuta un programa posible, en la máquina  $M_1$ , el trabajo  $J_1$  seguido del trabajo  $J_2$ , y en la máquina  $M_2$ , ejecuta el trabajo  $J_4$  seguido del trabajo  $J_3$ . Para esta programación,  $C_1 = 2, C_2 = 14, C_3 = 9, C_4 = 5$  y  $C_{\max} = 14$ . Una programación óptima ejecuta  $J_2$  en la máquina  $M_1$  y ejecuta los trabajos  $J_1, J_3$  y  $J_4$  en la máquina  $M_2$ . Para este programa,  $C_1 = 2, C_2 = 12, C_3 = 6, C_4 = 11$  y  $C_{\max} = 12$ .

Dado un problema de programación de máquinas paralelas, hacemos  $C_{\max}$  denota la marca de una programación óptima.

a. Muestre que el intervalo de procesamiento óptimo es al menos tan grande como el mayor procesamiento tiempo, es decir,

$$C_{\max} \quad \text{paquete} \\ \text{máximo : } 1kn$$

b. Muestre que el intervalo de producción óptimo es al menos tan grande como la carga promedio de la máquina, eso es,

$$C_{\max} \quad \frac{1}{\text{metro}} \quad X \quad \text{paquete : } \\ 1kn$$

Suponga que usamos el siguiente algoritmo codicioso para la programación de máquinas en paralelo: siempre que una máquina esté inactiva, programe cualquier trabajo que aún no se haya programado.

C. Escriba pseudocódigo para implementar este algoritmo codicioso. ¿Cuál es el tiempo de ejecución de su algoritmo?

d. Para el programa devuelto por el algoritmo codicioso, demuestre que



Concluya que este algoritmo es un algoritmo de 2 aproximaciones en tiempo polinomial.

35-6 Aproximación de un árbol generador máximo Sea GD .V;

E/ sea un grafo no dirigido con distintos pesos de arista  $w_{u,v}$ ; / en cada borde  $.u,v$ ; / 2 E. Para cada vértice  $V$  let  $\max_{u \in V} w_{u,v}$ ; / 2E  $f_{u,v}$ ; / g sea la arista de peso máximo que incide en ese vértice. Sea SG D  $f_{\max}$ ; / W 2 V g el conjunto de aristas de máximo peso que inciden en cada vértice, y sea TG el árbol generador de máximo peso de G, es decir, el árbol generador de máximo peso total. Para cualquier subconjunto de aristas  $E_0 \subseteq E$ , defina  $w(E_0) / DP$

$$w(E_0) = \sum_{(u,v) \in E_0} w_{u,v}$$

a. Da un ejemplo de un gráfico con al menos 4 vértices para el cual SG  $\neq$  TG.

b. Da un ejemplo de un gráfico con al menos 4 vértices para el cual SG  $\neq$  TG.

C. Demuestre que SG  $\leq$  TG para cualquier grafo G.

d. Demuestre que  $w(TG) / w(SG) \geq 2$  para cualquier gráfico G.

mi. Proporcione un algoritmo OV CE/-tiempo para calcular una aproximación de 2 al máximo peso de expansión de mamá.

35-7 Algoritmo de aproximación para el problema de la mochila 0-1 Recuerde el problema de la mochila de la sección 16.2. Hay  $n$  artículos, donde el  $i$ -ésimo artículo vale dólares y pesa  $w_i$  libras. También nos dan una mochila que puede contener como máximo  $W$  libras. Aquí, agregamos las suposiciones adicionales de que cada peso  $w_i$  es como máximo  $W$  y que los elementos están indexados en orden monótonamente decreciente de sus valores:

1            2            norte.

En el problema de la mochila 0-1, deseamos encontrar un subconjunto de artículos cuyo peso total sea como máximo  $W$  y cuyo valor total sea máximo. El problema de la mochila fraccionada es como el problema de la mochila 0-1, excepto que se nos permite tomar una fracción de cada artículo, en lugar de estar restringidos a tomar todos o ninguno de ellos.

Cada artículo. Si tomamos una fracción  $x_i$  del ítem  $i$ , donde  $0 \leq x_i \leq 1$ , aportamos  $x_i w_i$  al peso de la mochila y recibimos el valor  $x_i v_i$ . Nuestro objetivo es desarrollar un algoritmo de aproximación 2 en tiempo polinomial para el problema de la mochila 0-1.

Para diseñar un algoritmo de tiempo polinomial, consideraremos instancias restringidas del problema de la mochila 0-1. Dada una instancia  $I$  del problema de la mochila, formamos instancias restringidas  $I_j$ , para  $j \in \{1, 2, \dots, n\}$ , eliminando los elementos  $\{1, 2, \dots, j-1\}$  y requiere que la solución incluya el elemento  $j$  (todo el elemento  $j$  en los problemas fraccionarios y de mochila 0-1). No se eliminan elementos en la instancia  $I_1$ . Por ejemplo,  $I_j$  sea  $P_j$  una solución óptima al problema 0-1 y  $Q_j$  una solución óptima al problema fraccionario.

- a. Argumente que una solución óptima para la instancia  $I$  del problema de la mochila 0-1 es una de  $fP_1; P_2; \dots; P_n$ .
- b. Demuestre que podemos encontrar una solución óptima  $Q_j$  al problema fraccionario para el caso  $I_j$  al incluir el elemento  $j$  y luego usar el algoritmo codicioso en el que en cada paso, tomamos la mayor cantidad posible del elemento no elegido en el conjunto  $f_j \subset C_1; j \in C_2; \dots; n$  con valor máximo por libra  $i=w_i$ .
- c. Demuestre que siempre podemos construir una solución óptima  $Q_j$  para el problema fraccionario, por ejemplo  $I_j$ , que incluye como máximo un elemento fraccionariamente. Es decir, para todos los artículos excepto posiblemente uno, incluimos todo el artículo o ninguno en la mochila.
- d. Dada una solución óptima  $Q_j$  al problema fraccionario, por ejemplo  $I_j$ , forme la solución  $R_j$  a partir de  $Q_j$  eliminando cualquier elemento fraccionario de  $Q_j$ . Sea  $S$  el valor total de los elementos tomados en una solución  $S$ . Demuestre que  $R_j / S = Q_j$ .
- e. Proporcione un algoritmo de tiempo polinomial que devuelva una solución de valor máximo del conjunto  $fR_1; R_2; \dots; R_n$ , y demuestre que su algoritmo es un algoritmo de aproximación 2 en tiempo polinomial para el problema de la mochila 0-1.

#### Notas del capítulo

Aunque los métodos que no necesariamente calculan soluciones exactas se conocen desde hace miles de años (por ejemplo, métodos para aproximar el valor de  $\pi$ ), la noción de un algoritmo de aproximación es mucho más reciente. Hochbaum [172] acredita a Garey, Graham y Ullman [128] y Johnson [190] por formalizar el concepto de un algoritmo de aproximación de tiempo polinomial. El primer algoritmo de este tipo se atribuye a menudo a Graham [149].

Desde este trabajo inicial, se han diseñado miles de algoritmos de aproximación para una amplia gama de problemas, y existe una gran cantidad de literatura en este campo. Textos recientes de Ausiello et al. [26], Hochbaum [172] y Vazirani [345] se ocupan exclusivamente de algoritmos de aproximación, al igual que las encuestas de Shmoys [315] y Klein y Young [207]. Varios otros textos, como Garey y Johnson [129] y Papadimitriou y Steiglitz [271], también tienen una cobertura significativa de los algoritmos de aproximación. Lawler, Lenstra, Rinnooy Kan y Shmoys [225] brindan un extenso tratamiento de los algoritmos de aproximación para el problema del viajante de comercio.

Papadimitriou y Steiglitz atribuyen el algoritmo APPROX-VERTEX-COVER a F. Gavril y M. Yannakakis. El problema de la cobertura de vértices se ha estudiado extensamente (Hochbaum [172] enumera 16 algoritmos de aproximación diferentes para este problema), pero todas las proporciones de aproximación son al menos  $2 \text{ o } 1/$ .

El algoritmo APPROX-TSP-TOUR aparece en un artículo de Rosenkrantz, Stearns y Lewis [298]. Christofides mejoró este algoritmo y dio un algoritmo de aproximación de  $3 = 2$  para el problema del vendedor ambulante con la desigualdad del triángulo.

Arora [22] y Mitchell [257] han demostrado que si los puntos están en el plano euclidiano, existe un esquema de aproximación en tiempo polinomial. El teorema 35.3 se debe a Sahni y Gonzalez [301].

El análisis de la heurística codiciosa para el problema de cobertura de conjuntos está modelado a partir de la demostración publicada por Chv'atal [68] de un resultado más general; el resultado básico que se presenta aquí se debe a Johnson [190] y Lov'asz [238].

El algoritmo APPROX-SUBSET-SUM y su análisis están modelados libremente a partir de algoritmos de aproximación relacionados para los problemas de mochila y suma de subconjuntos de Ibarra y Kim [187].

El problema 35-7 es una versión combinatoria de un resultado más general en aproximadamente enteros de tipo mochila por Bienstock y McClosky [45].

El algoritmo aleatorio para la satisfacibilidad de MAX-3-CNF está implícito en el trabajo de Johnson [190]. El algoritmo ponderado de cobertura de vértices es de Hochbaum [171].

La Sección 35.4 solo toca el poder de la aleatorización y la programación lineal en el diseño de algoritmos de aproximación. Una combinación de estas dos ideas produce una técnica llamada "redondeo aleatorio", que formula un problema como un programa lineal entero, resuelve la relajación de programación lineal e interpreta las variables en la solución como probabilidades. Estas probabilidades luego ayudan a guiar la solución del problema original. Esta técnica fue utilizada por primera vez por Raghavan y Thompson [290] y ha tenido muchos usos posteriores. (Ver Motwani, Naor y Raghavan [261] para una encuesta.) Varias otras ideas recientes notables en el campo de los algoritmos de aproximación incluyen el método primal-dual (ver Goemans y Williamson [135] para una encuesta), encontrar cortes dispersos para uso en algoritmos divide y vencerás [229], y el uso de programación semidefinida [134].

Como se mencionó en las notas del capítulo 34, los resultados recientes en demostraciones comprobables probabilísticamente han llevado a límites inferiores en la aproximabilidad de muchos problemas, incluidos varios en este capítulo. Además de las referencias allí, el capítulo de Arora y Lund [23] contiene una buena descripción de la relación entre las pruebas verificables probabilísticamente y la dificultad de aproximar varios problemas.



---

## VIII Apéndice: Antecedentes matemáticos

---

## Introducción

Cuando analizamos algoritmos, a menudo necesitamos recurrir a un conjunto de herramientas matemáticas. Algunas de estas herramientas son tan simples como el álgebra de la escuela secundaria, pero otras pueden ser nuevas para usted. En la Parte I, vimos cómo manipular notaciones asintóticas y resolver recurrencias. Este apéndice comprende un compendio de varios otros conceptos y métodos que usamos para analizar algoritmos. Como se señaló en la introducción a la Parte I, es posible que haya visto gran parte del material de este apéndice antes de haber leído este libro (aunque las convenciones de notación específicas que usamos pueden diferir ocasionalmente de las que ha visto en otros lugares). Por lo tanto, debe tratar este apéndice como material de referencia. Sin embargo, como en el resto de este libro, hemos incluido ejercicios y problemas para que pueda mejorar sus habilidades en estas áreas.

El Apéndice A ofrece métodos para evaluar y acotar sumas, que ocurren con frecuencia en el análisis de algoritmos. Muchas de las fórmulas aquí aparecen en cualquier texto de cálculo, pero le resultará conveniente tener estos métodos compilados en un solo lugar.

El Apéndice B contiene definiciones y notaciones básicas para conjuntos, relaciones, funciones, gráficos y árboles. También da algunas propiedades básicas de estos objetos matemáticos.

El apéndice C comienza con los principios elementales del conteo: permutaciones, combinaciones y similares. El resto contiene definiciones y propiedades de probabilidad básica. La mayoría de los algoritmos de este libro no requieren probabilidad para su análisis y, por lo tanto, puede omitir fácilmente las últimas secciones del capítulo en una primera lectura, incluso sin hojearlas. Más tarde, cuando encuentre un análisis probabilístico que desee comprender mejor, encontrará el Apéndice C bien organizado para fines de referencia.

El apéndice D define las matrices, sus operaciones y algunas de sus propiedades básicas. Probablemente ya haya visto la mayor parte de este material si ha tomado un curso de álgebra lineal, pero puede resultarle útil tener un lugar para buscar nuestra notación y definiciones.

# A sumatorias

Cuando un algoritmo contiene una construcción de control iterativo, como un ciclo while o for , podemos expresar su tiempo de ejecución como la suma de los tiempos empleados en cada ejecución del cuerpo del ciclo. Por ejemplo, encontramos en la Sección 2.2 que la j-ésima iteración del ordenamiento por inserción tomó un tiempo proporcional  $a_j$  en el peor de los casos. Sumando el tiempo empleado en cada iteración, obtuvimos la sumatoria (o serie)

$\sum_{j=0}^n a_j$ :

Cuando evaluamos esta suma, alcanzamos un límite de  $n^2$  en el peor de los casos de tiempo de ejecución del algoritmo. Este ejemplo ilustra por qué debería saber cómo manipular y vincular sumas.

La sección A.1 enumera varias fórmulas básicas que involucran sumas. La Sección A.2 ofrece técnicas útiles para delimitar sumas. Presentamos las fórmulas en la Sección A.1 sin demostración, aunque las demostraciones de algunas de ellas aparecen en la Sección A.2 para ilustrar los métodos de esa sección. Puede encontrar la mayoría de las otras demostraciones en cualquier texto de cálculo.

## A.1 Fórmulas de suma y propiedades

Dada una secuencia  $a_1; a_2; \dots; a_n$  de números, donde  $n$  es un entero no negativo, podemos escribir la suma finita  $a_1 + a_2 + \dots + a_n$  como

$\sum_{k=1}^n a_k$  :

Si  $n = 0$ , el valor de la suma se define como 0. El valor de una serie finita siempre está bien definido y podemos sumar sus términos en cualquier orden.

Dada una secuencia infinita  $a_1; a_2; \dots$  de números, podemos escribir la suma infinita  $a_1 + a_2 + \dots$  como

$$\sum_{k=1}^{\infty} a_k$$

que interpretamos que significa

$$\lim_{n \rightarrow \infty} \sum_{k=1}^n a_k$$

Si el límite no existe, la serie diverge; de lo contrario, converge. Los términos de una serie convergente no siempre se pueden sumar en ningún orden. Sin embargo, podemos reordenar los términos de una serie absolutamente convergente, es decir, una serie  $\sum_{k=1}^{\infty} a_k$  para la cual la serie  $\sum_{k=1}^{\infty} |a_k|$  también converge.

linealidad

Para cualquier número real  $c$  y cualquier secuencia finita  $a_1; a_2; \dots; a_n$  y  $b_1; b_2; \dots; b_n$ ,

$$\sum_{k=1}^n c a_k = c \sum_{k=1}^n a_k$$

La propiedad de linealidad también se aplica a series convergentes infinitas.

Podemos explotar la propiedad de linealidad para manipular sumas incorporando notación asintótica. Por ejemplo,

$$\sum_{k=1}^n f(k) = \sum_{k=1}^{\infty} f(k)$$

En esta ecuación, la notación , en el lado izquierdo se aplica a la variable  $k$ , pero en el lado derecho se aplica a  $n$ . También podemos aplicar tales manipulaciones a series convergentes infinitas.

Serie aritmética

la suma

$$\sum_{k=1}^{\infty} k = \frac{1}{2} n(n+1)$$

es una serie aritmética y tiene el valor

$$\sum_{k=1}^{\infty} k = \frac{1}{2} n(n+1) \quad (A.1)$$

$$D = \frac{1}{2} n(n+1) \quad (A.2)$$

## Sumas de cuadrados y cubos

Tenemos las siguientes sumas de cuadrados y cubos:

$$\sum_{k=0}^n k^2 = \frac{n(n+1)(2n+1)}{6}; \quad (\text{A.3})$$

$$\sum_{k=0}^n k^3 = \frac{n^2(n+1)^2}{4}; \quad (\text{A.4})$$

## Series geométricas

Para real  $x \neq 1$ , la sumatoria

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

es una serie geométrica o exponencial y tiene el valor

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}; \quad (\text{A.5})$$

Cuando la sumatoria es infinita y  $|x| < 1$ , tenemos la serie geométrica decreciente infinita

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}; \quad (\text{A.6})$$

## Serie armónica

Para números enteros positivos  $n$ , el  $n$ -ésimo número armónico es

$$\begin{aligned} H_n &= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}, \\ \sum_{k=1}^n \frac{1}{k} &= H_n - H_{n-1} \end{aligned} \quad (\text{A.7})$$

(Probaremos un límite relacionado en la Sección A.2.)

## Integrando y diferenciando series.

Al integrar o diferenciar las fórmulas anteriores, surgen fórmulas adicionales. Por ejemplo, derivando ambos lados de la serie geométrica infinita (A.6) y multiplicando por  $x$ , obtenemos

$$\frac{x_1 - x_k}{k-1} \leq \frac{x}{2} \quad (A.8)$$

para  $|x| < 1$ .

### Serie telescópica

Para cualquier secuencia  $a_0; a_1; \dots; a_n$ ,

$$\sum_{k=1}^n (a_k - a_{k-1}) = a_n - a_0 \quad (A.9)$$

ya que cada uno de los términos  $a_1; a_2; \dots; a_n$  se suma exactamente una vez y se resta exactamente una vez. Decimos que la suma se eleva. Similarmente,

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n$$

Como ejemplo de una suma telescópica, considere la serie

$$\sum_{k=1}^{\infty} \frac{1}{k(k+1)}$$

Como podemos reescribir cada término como

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$$

obtenemos

$$\begin{aligned} \sum_{k=1}^{\infty} \frac{1}{k(k+1)} &= \left( \frac{1}{1} - \frac{1}{2} \right) + \left( \frac{1}{2} - \frac{1}{3} \right) + \dots \\ &= 1 - \frac{1}{\infty} \end{aligned}$$

### Productos

Podemos escribir el producto finito  $a_1 a_2 \dots a_n$  como

$$\prod_{k=1}^n a_k$$

Si  $n = 0$ , el valor del producto se define como 1. Podemos convertir una fórmula con un producto en una fórmula con una suma usando la identidad

$$\lg \prod_{k=1}^n a_k = \sum_{k=1}^n \lg a_k$$

## Ejercicios

A.1-1

Encuentre una fórmula simple para  $P_n \sum_{k=1}^n k^{-1}$ .

A.1-2 ?

Demuestre que  $\prod_{k=1}^n k^{-1} = \frac{1}{n!} \ln(n+1) - \gamma$  manipulando la serie armónica.

A.1-3

Demuestre que  $\prod_{k=1}^n k^{2x} \leq e^{x^2}$  para  $0 < x < 1$ .

A.1-4 ?

Demuestre que  $P_1 \sum_{k=0}^{\infty} k^{-1} = \infty$ .

A.1-5 ?

Evalúa la suma  $P_1 \sum_{k=1}^{\infty} k^{-2}$ .

A.1-6

Demuestre que  $\sum_{k=1}^{\infty} k^{-1} \ln(k) \geq \ln(2)$  usando la propiedad de linealidad de las sumatorias.

A.1-7

Evaluar el producto  $Q_n = \prod_{k=1}^n k^{-2}$ .

A.1-8 ?

Evaluar el producto  $Q_n \sum_{k=1}^{\infty} k^{-1}$ .**A.2 Sumatorias acotadas**

Tenemos muchas técnicas a nuestra disposición para acotar las sumas que describen los tiempos de ejecución de los algoritmos. Estos son algunos de los métodos más utilizados.

## Inducción matemática

La forma más básica de evaluar una serie es usar la inducción matemática. Como ejemplo, demostremos que la serie aritmética  $P_n n(n+1)/2$ . Podemos ~~verificar fácilmente~~ <sup>1</sup> esta afirmación para  $n \leq 2$ . Hacemos la suposición inductiva de que

se cumple para  $n$ , y probamos que se cumple para  $n + 1$ . Tenemos

$$\frac{X_n C_1}{k D_1} \leq \frac{k C_1}{k D_1}$$

$$\stackrel{D}{\rightarrow} \frac{n C_1}{k D_1} \leq \frac{(n+1) C_1}{k D_1}$$

$$\stackrel{D}{\rightarrow} \frac{C_1}{k D_1} \leq \frac{C_1}{k D_1}$$

No siempre es necesario adivinar el valor exacto de una suma para utilizar la inducción matemática. En su lugar, puede usar la inducción para probar un límite en una suma. Como ejemplo, demostremos que la serie geométrica  $P_n$  más específicamente, demostremos que  $\lim_{n \rightarrow \infty} P_n = 0$ . Como ejemplo, demostremos que la serie geométrica  $P_n$  más específicamente, demostremos que  $\lim_{n \rightarrow \infty} P_n = 0$ .

que la condición inicial  $P_0 = 1$ , tenemos  $P_0 = 1$  para alguna constante  $c$ . Para el

$$\frac{P_0}{k D_0} = \frac{1}{3} < 1$$

que el límite se cumple para  $n$ , demostremos que se cumple para  $n + 1$ . Tenemos

$$\frac{X_n C_1}{k D_0} \leq \frac{3^n C_1}{k D_0}$$

$c 3^n C_1 \leq 3^n C_1$  (por la hipótesis inductiva)

$$\stackrel{D}{\rightarrow} \frac{1}{c} \leq \frac{1}{c} c 3^n C_1$$

$$3^n C_1$$

siempre que  $c < 1$ , de manera equivalente,  $c < 1$ . Por lo tanto,  $\lim_{n \rightarrow \infty} P_n = 0$ .

Debemos tener cuidado cuando usamos la notación asintótica para probar los límites en  $k D$ . Considera la siguiente prueba falaz de que  $P_n = 1$ . Ciertamente,  $\lim_{n \rightarrow \infty} P_n = 1$ . Suponiendo que el límite se cumple para  $n$ , ahora lo demostramos para  $n + 1$ :

$$\frac{X_n C_1}{k D_1} \leq \frac{k C_1}{k D_1}$$

$$\stackrel{D}{\rightarrow} \frac{C_1}{k D_1} \leq \frac{C_1}{k D_1}$$

$$\stackrel{D}{\rightarrow} C_1 \leq C_1$$

El error en el argumento es que la "constante" oculta por el "gran oh" crece con  $n$  y, por lo tanto, no es constante. No hemos demostrado que la misma constante funcione para todo  $n$ .

Limitando los términos

A veces podemos obtener un buen límite superior de una serie al acotar cada término de la serie y, a menudo, basta con usar el término más grande para acotar los demás. Para

ejemplo, un límite superior rápido en la serie aritmética (A.1) es

$$\frac{X_n}{kD1} \leq \frac{X_n}{kD1} = n^2$$

En general, para una serie  $\sum_{k=0}^{\infty} ak$ , si hacemos  $a_{\max} D \geq \max_{k \geq 0} |ak|$ , entonces

$$\frac{X_n}{kD1} \leq a_{\max}$$

La técnica de acotar cada término de una serie por el término más grande es un método débil cuando la serie puede, de hecho, estar acotada por una serie geométrica. Dada la serie  $\sum_{k=0}^{\infty} ak$ , suponga que  $|ak| \leq |a_0|r^k$  para todo  $k \geq 0$ , donde  $0 < r < 1$  es una constante. Podemos acotar la suma por una serie geométrica decreciente infinita, ya que  $|a_0|r^k$ , y así

$$\begin{aligned} \frac{X_n}{kD0} &\leq \frac{X_1}{kD0} a_0 r^k \\ &\leq \frac{a_0}{1-r} \end{aligned}$$

Podemos aplicar este método para acotar la sumatoria  $\sum_{k=1}^{\infty} kC_1$ . Para comenzar la suma en  $k=0$ , lo reescribimos como  $\sum_{k=0}^{\infty} kC_1 = C_1 + \sum_{k=1}^{\infty} kC_1$ . El primer término ( $a_0$ ) es  $C_1 = 1$ , y la razón ( $r$ ) de términos consecutivos es

$$\begin{aligned} \frac{C_2}{C_1} &= 3 \\ \frac{C_3}{C_2} &= 3 \\ \vdots & \vdots \\ \frac{C_k}{C_{k-1}} &= 3 \end{aligned}$$

para todo  $k \geq 1$ . Así, tenemos

$$\begin{aligned} \sum_{k=1}^{\infty} kC_1 &= C_1 + \sum_{k=1}^{\infty} kC_1 \\ &= 1 + \frac{1}{1-3} \\ &= \frac{1}{2} \end{aligned}$$

Un error común al aplicar este método es mostrar que la razón de términos consecutivos es menor que 1 y luego suponer que la suma está limitada por una serie geométrica. Un ejemplo es la serie armónica infinita, que diverge ya que

$$\frac{1}{k} \underset{n \rightarrow \infty}{\lim} \frac{x_n}{x_{n-1}} = \frac{1}{k} \underset{n \rightarrow \infty}{\lim} \frac{\lg n}{\lg(n-1)} = 1$$

La razón de los términos  $\frac{x_n}{x_{n-1}}$  en esta serie es  $k = \frac{1}{k} < 1$ , pero la serie no está limitada por una serie geométrica decreciente. Para acotar una serie por una serie geométrica, debemos demostrar que existe un  $r < 1$ , que es una constante, tal que la razón de todos los pares de términos consecutivos nunca excede de  $r$ . En la serie armónica, tal  $r$  no existe porque la relación se vuelve arbitrariamente cercana a 1.

### División de sumas

Una forma de obtener límites en una suma difícil es expresar la serie como la suma de dos o más series dividiendo el rango del índice y luego acotar cada una de las series resultantes. Por ejemplo, supongamos que tratamos de encontrar un límite inferior en la serie aritmética  $\sum_{k=1}^n k$ , que ~~ya~~ hemos visto que tiene un límite superior de  $n^2$ . Podríamos intentar acotar cada término en la suma por el término más pequeño, pero dado que ese término es 1, obtenemos un límite inferior de  $n$  para la suma, muy lejos de nuestro límite superior de  $n^2$ .

Podemos obtener un mejor límite inferior dividiendo primero la suma. Asumir por conveniencia de que  $n$  sea par. Tenemos

$$\begin{aligned} n &= 2 \\ \frac{x_n}{k} &\leq \frac{x_{n/2}}{k/2} & k \\ \frac{x_{n/2}}{k/2} &\leq \frac{x_1}{1} & k = 2 \\ \frac{x_1}{1} &\leq \frac{x_{n/2}}{k/2} & n/2 \\ D \cdot n/2 &\leq x_n & n/2 \\ D \cdot n/2 &\leq x_n & ; \end{aligned}$$

que es un límite asintóticamente estrecho, ya que  $\frac{x_{n/2}}{k/2} \leq O(n/2)$ .

Para una suma que surge del análisis de un algoritmo, a menudo podemos dividir la suma e ignorar un número constante de términos iniciales. Generalmente, esta técnica se aplica cuando cada término  $a_k$  en una suma  $\sum_{k=0}^{n-1} a_k$  es independiente de  $n$ .

Entonces, para cualquier constante  $k_0 > 0$ , podemos escribir

$$\begin{array}{l} k \\ \frac{X_n}{kD_0} \leq \frac{X_{01}}{kD_0} + C \frac{X_n}{kDk_0} \\ \text{DO.1/C } X_n \leq \frac{ak}{kDk_0}; \end{array}$$

dado que los términos iniciales de la suma son todos constantes y hay una constante numero de ellos Entonces podemos usar otros métodos para enlazar  $\sum P_n$   $kDk_0$   $ak$ . Esta técnica también se aplica a sumas infinitas. Por ejemplo, para encontrar un límite superior asintótico en

$$\frac{k^2}{X_1 \frac{2k}{kD_0}}$$

observamos que la razón de términos consecutivos es

$$\frac{\frac{.kC}{1/2}=2kC_1}{k^2=2k} = \frac{\frac{.kC}{1/2}}{2k^2} = \frac{8}{9}$$

si  $k > 3$ . Por lo tanto, la suma se puede dividir en

$$\begin{array}{l} \frac{k^2}{X_1 \frac{2k}{kD_0}} \text{ DO. } \frac{k^2}{kD_0} \text{ CX } \frac{k^2}{kD_3} \\ \frac{2}{X_1 \frac{2k}{kD_0}} \text{ C } \frac{9}{8} \frac{X_1}{kD_0} \frac{8}{9}^k \\ \text{DO.1/;} \end{array}$$

ya que la primera sumatoria tiene un número constante de términos y la segunda sumatoria es una serie geométrica decreciente.

La técnica de división de sumas puede ayudarnos a determinar límites asintóticos en situaciones mucho más difíciles. Por ejemplo, podemos obtener un límite de  $O.lg n$  en la serie armónica (A.7):

$$\frac{1}{H_n D X_n \frac{1}{k}}$$

Lo hacemos dividiendo el rango de  $1 \leq n \leq b$  en  $c$  partes y limitando la contribución de cada parte por 1. Para  $i \in \{0, 1, \dots, b\}$ , la  $i$ -ésima pieza consiste

de los términos que empiezan en  $1=2i$  y van hasta  $1=2iC_1$ , pero sin incluirlo. La última pieza podría contener términos que no están en la serie armónica original y, por lo tanto, tenemos

$$\begin{aligned}
 & X_n \frac{1}{k} \sum_{i=D_0}^{b \lg n c} \frac{1}{2X_i 1 \frac{1}{2i C_j}} \\
 & \sum_{i=D_0}^{b \lg n c} \frac{1}{2X_i 1 \frac{1}{2i}} \\
 & \sum_{i=D_0}^{b \lg n c} \frac{1}{2X_i} \\
 & \ln n C_1
 \end{aligned} \tag{A.10}$$

### Aproximación por integrales

Cuando una suma tiene la forma  $\sum_{k=D_m}^{nC_1} f . k / Z$ , donde  $f . k / Z$  es monótonamente función de plegado, podemos aproximarla por integrales:

$$\sum_{k=D_m}^{nC_1} f . k / Z \approx \int_{m1}^{nC_1} f . x / dx \tag{A.11}$$

La figura A.1 justifica esta aproximación. La suma se representa como el área de los rectángulos en la figura y la integral es la región sombreada debajo de la curva. Cuando  $f . k / Z$  es una función monótonamente decreciente, podemos usar un método similar para proporcionar los límites

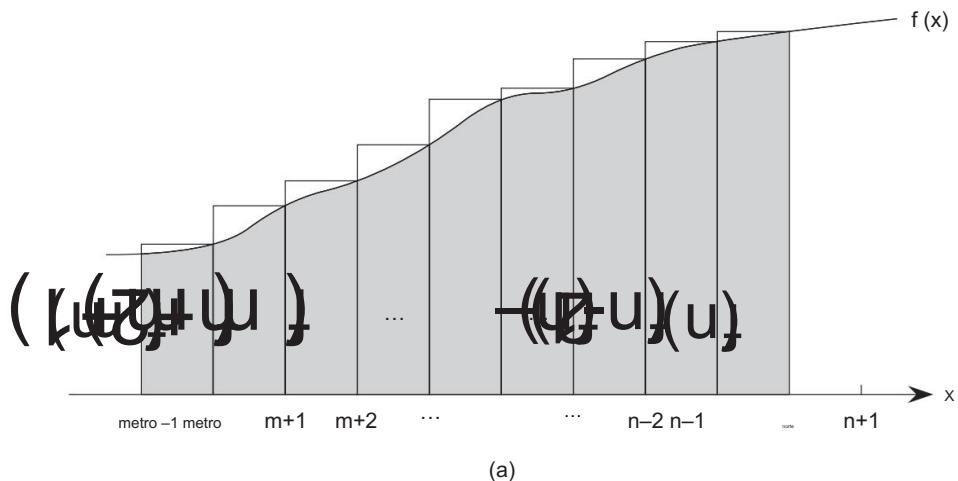
$$\sum_{k=D_m}^{nC_1} f . k / Z \approx \int_{m1}^{nC_1} f . x / dx \tag{A.12}$$

La aproximación integral (A.12) da una estimación ajustada para el  $n$ -ésimo armónico número. Para un límite inferior, obtenemos

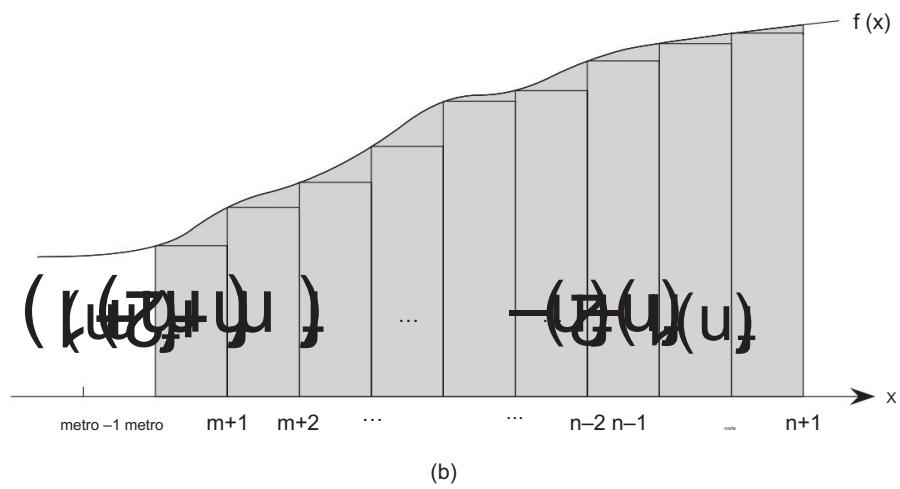
$$X_n \frac{1}{k} Z \approx \int_1^{nC_1} \frac{dx}{x} = \ln(nC_1) \tag{A.13}$$

Para el límite superior, derivamos la desigualdad

$$X_n \frac{1}{k} Z \approx \int_1^{nC_1} \frac{dx}{x} = \ln(nC_1)$$



(a)



(b)

Figura A.1 Aproximación de  $\int_a^b f(x) dx$  por integrales. El área de cada rectángulo se muestra dentro del rectángulo y el área total del rectángulo representa el valor de la suma. La integral está representada por el área sombreada bajo la curva. Al comparar áreas en (a), obtenemos  $R$

$$\int_a^b f(x) dx \approx \sum_{k=1}^{n-1} f(x_k) \Delta x$$

y luego desplazando los rectángulos una unidad a la derecha, obtenemos

$$\int_a^b f(x) dx \approx \sum_{k=0}^{n-1} f(x_k) \Delta x$$

que produce el límite

$$\lim_{n \rightarrow \infty} \frac{1}{\ln n} = 0 \quad \text{en } n \geq 1 \quad \text{(A.14)}$$

### Ejercicios

#### A.2-1

Demuestre que  $\sum_{k=1}^{\infty} \frac{1}{k^2}$  está acotado arriba por una constante.

#### A.2-2

Encuentre un límite superior asintótico en la sumatoria

$b \ln n$

$$\sum_{k=0}^{n-1} \frac{1}{k^2} \approx \int_0^n \frac{1}{x^2} dx = -\frac{1}{x} \Big|_0^n = \frac{1}{n}$$

#### A.2-3

Muestre que el número armónico  $n$ -ésimo es  $\ln n + \gamma$  dividiendo la suma.

#### A.2-4

Pruebe que  $\sum_{k=1}^{\infty} \frac{1}{k^3}$  converge con una integral.

#### A.2-5

¿Por qué no usamos la aproximación integral (A.12) directamente en  $P_n$  para  $\sum_{k=1}^n \frac{1}{k} = H_n$  a obtener un límite superior en el  $n$ -ésimo número armónico?

### Problemas

#### A-1 Sumatorios acotados

Proporcione límites asintóticamente estrechos en los siguientes sumatorios. Suponga que  $r > 0$  y  $s > 0$  son constantes.

a.  $\sum_{k=1}^{\infty} \frac{k^r}{k^s}$

b.  $\sum_{k=1}^{\infty} \frac{\ln k}{k^s}$

### C. $\sum_{k=1}^n k r \lg k$ .

kD1

---

#### Notas del apéndice

Knuth [209] proporciona una excelente referencia para el material presentado aquí. Puede encontrar las propiedades básicas de las series en cualquier buen libro de cálculo, como Apostol [18] o Thomas et al. [334].

**B****Conjuntos, Etc**

Muchos capítulos de este libro tocan los elementos de las matemáticas discretas. Este apéndice repasa más completamente las notaciones, definiciones y propiedades elementales de conjuntos, relaciones, funciones, gráficas y árboles. Si ya está bien versado en este material, probablemente pueda hojear este capítulo.

**B.1 Conjuntos**

Un conjunto es una colección de objetos distinguibles, llamados sus miembros o elementos. Si un objeto  $x$  es miembro de un conjunto  $S$ , escribimos  $x \in S$  (léase “ $x$  es miembro de  $S$ ” o, más brevemente, “ $x$  está en  $S$ ”). Si  $x$  no es miembro de  $S$ , escribimos  $x \notin S$ . Podemos describir un conjunto enumerando explícitamente sus miembros como una lista entre llaves. Por ejemplo, podemos definir un conjunto  $S$  para que contenga precisamente los números 1, 2 y 3 escribiendo  $S = \{1; 2; 3\}$ . Como 2 es miembro del conjunto  $S$ , podemos escribir  $2 \in S$ , y como 4 no es miembro, tenemos  $4 \notin S$ . Un conjunto no puede contener el mismo objeto más de una vez,<sup>1</sup> y sus elementos no están ordenados. Dos conjuntos  $A$  y  $B$  son iguales, escritos  $A = B$ , si contienen los mismos elementos. Por ejemplo,  $\{1; 2; 3\} = \{3; 2; 1\}$ .

Adoptamos notaciones especiales para conjuntos que se encuentran con frecuencia:

$\emptyset$  denota el conjunto vacío, es decir, el conjunto que no contiene miembros.

$\mathbb{Z}$  denota el conjunto de enteros, es decir, el conjunto  $\{\dots; -2; -1; 0; 1; 2; \dots\}$ .

$\mathbb{R}$  denota el conjunto de números reales.

$\mathbb{N}$  denota el conjunto de números naturales, es decir, el conjunto  $\{0; 1; 2; \dots\}$ .

2

<sup>1</sup>Una variación de un conjunto, que puede contener el mismo objeto más de una vez, se denomina multiconjunto .

<sup>2</sup>Algunos autores comienzan los números naturales con 1 en lugar de 0. La tendencia moderna parece ser comenzar con 0.

Si todos los elementos de un conjunto A están contenidos en un conjunto B, es decir, si  $x \in A$  implica  $x \in B$ , entonces escribimos  $A \subseteq B$  y decimos que A es un subconjunto de B. Un conjunto A es un subconjunto propio de B , escrito  $A \subset B$ , si  $A \subseteq B$  pero  $A \neq B$ . (Algunos autores usan el símbolo " $\subset$ " para denotar la relación de subconjunto ordinaria, en lugar de la relación de subconjunto propia.) Para cualquier conjunto A, tenemos  $A \subseteq A$ . Para dos conjuntos A y B, tenemos  $A \subseteq B$  si y sólo si  $A \subseteq B$  y  $B \subseteq A$ . Para cualesquiera tres conjuntos A, B y C, si  $A \subseteq B$  y  $B \subseteq C$ , entonces  $A \subseteq C$ . Para cualquier conjunto A, tenemos  $\emptyset \subseteq A$ .

A veces definimos conjuntos en términos de otros conjuntos. Dado un conjunto A, podemos definir un conjunto  $B = \{x \in A \mid P(x)\}$  estableciendo una propiedad que distinga los elementos de B. Por ejemplo, podemos definir el conjunto de los enteros pares mediante  $\{x \in \mathbb{Z} \mid x = 2k \text{ para } k \in \mathbb{Z}\}$ . Los dos puntos en esta notación se leen como "tal que". (Algunos autores usan una barra vertical en lugar de los dos puntos).

Dados dos conjuntos A y B, también podemos definir nuevos conjuntos aplicando operaciones de conjunto:

La intersección de los conjuntos A y B es el conjunto

$A \cap B = \{x \in A \text{ y } x \in B\}$  :

La unión de los conjuntos A y B es el conjunto

$A \cup B = \{x \in A \text{ o } x \in B\}$  :

La diferencia entre dos conjuntos A y B es el conjunto

$A \setminus B = \{x \in A \text{ y } x \notin B\}$  :

Las operaciones con conjuntos obedecen las siguientes leyes:

Leyes de conjuntos vacíos:

$\emptyset' = \emptyset$  ;

$\emptyset'' = \emptyset$  ;

Leyes de idempotencia:

$\emptyset' = \emptyset$  ;

$\emptyset'' = \emptyset$  ;

Leyes conmutativas:

$\emptyset' \cup \emptyset'' = \emptyset'' \cup \emptyset'$  ;

$\emptyset' \cap \emptyset'' = \emptyset'' \cap \emptyset'$  ;

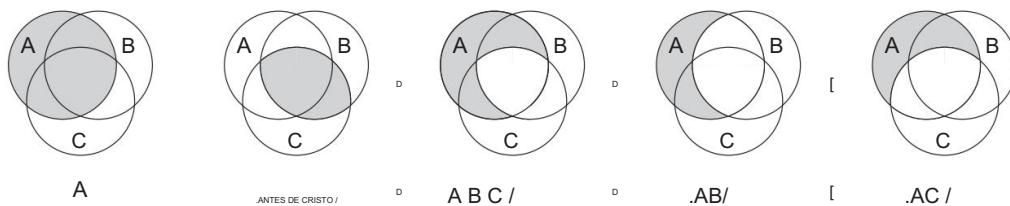


Figura B.1 Diagrama de Venn que ilustra la primera de las leyes de DeMorgan (B.2). Cada uno de los conjuntos A, B y C se representa como un círculo.

Leyes asociativas:

$$\begin{aligned} & A \setminus B \setminus C / D . A \setminus B / \setminus C A [ . B [ C / D . A \\ & [ B / [ C : \end{aligned}$$

Leyes distributivas:

$$\begin{aligned} & A \setminus B [ C / D . A \setminus B / [ . A \setminus C /; A [ . B \setminus C / D . A [ B / \\ & \setminus . A [ C /: \end{aligned} \tag{B.1}$$

Leyes de absorción:

$$\begin{aligned} & A \setminus A [ B / D A A [ . A \setminus B / \\ & D A \end{aligned}$$

Leyes de DeMorgan:

$$A . B \setminus C / D . A B / [ . A C /; A . B [ C / D . A B / \setminus . A C /: \tag{B.2}$$

La figura B.1 ilustra la primera de las leyes de DeMorgan, utilizando un diagrama de Venn: una imagen gráfica en la que los conjuntos se representan como regiones del plano.

A menudo, todos los conjuntos bajo consideración son subconjuntos de un conjunto U más grande llamado universo. Por ejemplo, si estamos considerando varios conjuntos formados únicamente por números enteros, el conjunto Z de números enteros es un universo apropiado. Dado un universo U, definimos el complemento de un conjunto A como  $\overline{A} = U \setminus A$ . Para cualquier conjunto AU, tenemos las siguientes leyes:

$$\begin{aligned} & \overline{\overline{A}} = A \\ & A \setminus \overline{A} = U \\ & A [ \overline{A} = \emptyset \end{aligned}$$

Podemos reescribir las leyes de DeMorgan (B.2) con complementos de conjuntos. Para dos conjuntos cualesquiera  $B; C$ , tenemos

$$\overline{B \setminus C} = \overline{B} \cup \overline{C} ;$$

$$B = \overline{\overline{B} \cap \overline{C}} ;$$

Dos conjuntos  $A$  y  $B$  son disjuntos si no tienen elementos en común, es decir, si  $A \cap B = \emptyset$ . Una colección  $S$  de conjuntos no vacíos forma una partición de un conjunto  $S$  si

los conjuntos son disjuntos por pares, es decir,  $S_i \cap S_j = \emptyset$  implica  $S_i \cup S_j = S$ ; y

su unión es  $S$ , es decir,

DAKOTA DEL SUR [ Sí :

$S = \bigcup_{i=1}^n S_i$

En otras palabras,  $S$  forma una partición de  $S$  si cada elemento de  $S$  aparece en exactamente un  $S_i$ .

El número de elementos en un conjunto es la cardinalidad (o tamaño) del conjunto, denotado  $|S|$ . Dos conjuntos tienen la misma cardinalidad si sus elementos se pueden poner en una correspondencia uno a uno. La cardinalidad del conjunto vacío es  $|\emptyset| = 0$ . Si la cardinalidad de un conjunto es un número natural, decimos que el conjunto es finito; de lo contrario, es infinito. Un conjunto infinito que se puede poner en una correspondencia biunívoca con los números naturales  $\mathbb{N}$  es contablemente infinito; de lo contrario, es incontable. Por ejemplo, los números enteros  $\mathbb{Z}$  son contables, pero los reales  $\mathbb{R}$  no son contables.

Para cualesquier dos conjuntos finitos  $A$  y  $B$ , tenemos la identidad

$$|A \cup B| = |A| + |B| - |A \cap B| ; \quad (B.3)$$

de lo cual podemos concluir que

$$|A \cup B| = |A| + |B| - |A \cap B| ;$$

Si  $A$  y  $B$  son disjuntos, entonces  $|A \cup B| = |A| + |B|$  y por lo tanto  $|A \cup B| = |A| + |B| - |A \cap B|$ . Si  $A \cap B = \emptyset$ , entonces  $|A \cup B| = |A| + |B|$ .

Un conjunto finito de  $n$  elementos a veces se denomina  $n$ -conjunto. Un conjunto  $1$  se llama único. Un subconjunto de  $k$  elementos de un conjunto a veces se denomina  $k$ -subconjunto.

Denotamos el conjunto de todos los subconjuntos de un conjunto  $S$ , incluido el conjunto vacío y  $S$  mismo, por  $2^S$ ; llamamos a  $2^S$  el conjunto potencia de  $S$ . Por ejemplo,  $\{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}$ . El conjunto potencia de un conjunto finito  $S$  tiene cardinalidad  $2^{|S|}$  (vea el Ejercicio B.1-5).

A veces nos interesan las estructuras en forma de conjunto en las que se ordenan los elementos.

Un par ordenado de dos elementos  $a$  y  $b$  se denota  $(a, b)$  y se define formalmente como el conjunto  $\{a, b\}$ . Así, el par ordenado  $(a, b)$  no es lo mismo que el par ordenado  $(b, a)$ .

El producto cartesiano de dos conjuntos A y B, denotado  $AB$ , es el conjunto de todos los pares ordenados tales que el primer elemento del par es un elemento de A y el segundo es un elemento de B. Más formalmente,

$\{AB\} = \{(a, b) | a \in A \text{ y } b \in B\}$

Por ejemplo,  $\{f\} = \{(a, f) | a \in A\}$ . Si A y B son conjuntos finitos, la cardinalidad de su producto cartesiano es

$$|A \times B| = |A| \cdot |B| \quad (B.4)$$

El producto cartesiano de n conjuntos  $A_1; A_2; \dots; A_n$  es el conjunto de n-tuplas

$$A_1 \times A_2 \times \dots \times A_n = \{(a_1, a_2, \dots, a_n) | a_i \in A_i \text{ para } i = 1, 2, \dots, n\}$$

cuya cardinalidad es

$$|A_1 \times A_2 \times \dots \times A_n| = |A_1| \cdot |A_2| \cdot \dots \cdot |A_n|$$

si todos los conjuntos son finitos. Denotamos un producto cartesiano de n veces sobre un solo conjunto A por el conjunto

$$A^n = A \times A \times \dots \times A \quad (n \text{ veces})$$

cuya cardinalidad es  $|A^n| = |A|^n$  si A es finito. También podemos ver una n-tupla como una secuencia finita de longitud n (ver página 1166).

### Ejercicios

#### B.1-1

Dibujar diagramas de Venn que ilustren la primera de las leyes distributivas (B.1).

#### B.1-2

Demostrar la generalización de las leyes de DeMorgan a cualquier colección finita de conjuntos:

$$\begin{array}{c} \overline{A_1 \setminus A_2 \cup \dots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \dots \cap \overline{A_n} \\ \overline{A_1 \cup A_2 \cup \dots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \dots \cap \overline{A_n} \end{array}$$

## B.1-3 ?

Demuestre la generalización de la ecuación (B.3), que se denomina principio de inclusión y exclusión:

$$jA1 \cup A2 \cup \dots \cup An \cup D \cup A1 \cup C$$

$$jA2 \cup CC \cup jAn \cup jA1 \cup A2 \cup jA1 \cup \dots$$

$$A3 \cup \dots$$

(todos los pares)

$$C \cup jA1 \cup A2 \cup \dots \cup A3 \cup C$$

(todos los triples)

⋮

$$C \cup \dots \cup jA1 \cup A2 \cup \dots \cup An :$$

## B.1-4

Demuestre que el conjunto de los números naturales impares es contable.

## B.1-5

Muestre que para cualquier conjunto finito S, el conjunto potencia  $2^S$  tiene  $2^{|S|}$  elementos (es decir, hay  $2^{|S|}$  subconjuntos distintos de S).

## B.1-6

Dé una definición inductiva para una n-tupla extendiendo la definición de la teoría de conjuntos para un par ordenado.

## B.2 Relaciones

Una relación binaria R sobre dos conjuntos A y B es un subconjunto del producto cartesiano AB. Si  $a; b \in R$ , a veces escribimos  $aRb$ . Cuando decimos que R es una relación binaria sobre un conjunto A, queremos decir que R es un subconjunto de  $A \times A$ . Por ejemplo, la relación "menor que" sobre los números naturales es el conjunto  $\{a; b \in \mathbb{N} \mid a < b\}$ . Una relación n-aria en los conjuntos  $A_1; A_2; \dots; A_n$  es un subconjunto de  $A_1 \times A_2 \times \dots \times A_n$ .

Una relación binaria RAA es reflexiva si

para todo  $a \in A$ . Por ejemplo, " $=$ " y " $\in$ " son relaciones reflexivas sobre  $\mathbb{N}$ , pero " $<$ " no lo es.

La relación R es simétrica si  $aRb$

implica  $bRa$  para todo

$a; b \in A$ . Por ejemplo, " $=$ " es simétrica, pero " $<$ " y " $\neq$ " no lo son. El

la relación R es transitiva si

$aRb$  y  $bRc$  implican  $aRc$

para todo  $a; b; c \in A$ . Por ejemplo, las relaciones " $<$ ", " $=$ " y " $\sim$ " son transitivas, pero la relación  $R$  no es transitiva, ya que  $3R4$  y  $4R5$  no implican  $3R5$ .

Una relación reflexiva, simétrica y transitiva es una relación de equivalencia.

Por ejemplo, " $\sim$ " es una relación de equivalencia sobre los números naturales, pero " $<$ " no lo es.

Si  $R$  es una relación de equivalencia sobre un conjunto  $A$ , entonces para  $a \in A$ , la clase de equivalencia de  $a$  es el conjunto  $\{x \in A \mid aRx\}$ , es decir, el conjunto de todos los elementos equivalentes a  $a$ .

Por ejemplo, si definimos  $R$  como  $R = \{(a, b) \in \mathbb{N}^2 \mid a \text{ es par} \wedge b \text{ es par}\}$ , entonces  $R$  es una relación de equivalencia, ya que  $a$  es par (reflexivo),  $a \sim b$  implica que  $b$  es par (simétrico) y  $a \sim b$  y  $b \sim c$  implica que  $a \sim c$  (transitivo). La clase de equivalencia de  $4$  es  $\{x \in \mathbb{N} \mid x \text{ es par}\}$ , y la clase de equivalencia de  $3$  es  $\{x \in \mathbb{N} \mid x \equiv 3 \pmod{2}\}$ . Un teorema básico de clases de equivalencia es el siguiente.

**Teorema B.1** (Una relación de equivalencia es lo mismo que una partición)

Las clases de equivalencia de cualquier relación de equivalencia  $R$  en un conjunto  $A$  forman una partición de  $A$ , y cualquier partición de  $A$  determina una relación de equivalencia en  $A$  para la cual los conjuntos en la partición son las clases de equivalencia.

**Demostración** Para la primera parte de la demostración, debemos mostrar que las clases de equivalencia de  $R$  son conjuntos disjuntos por pares no vacíos cuya unión es  $A$ . Como  $R$  es reflexivo,  $a \in \{x \in A \mid aRx\}$ , y por lo tanto las clases de equivalencia no son vacías; además, como todo elemento  $a \in A$  pertenece a la clase de equivalencia  $\{x \in A \mid aRx\}$ , la unión de las clases de equivalencia es  $A$ . Resta demostrar que las clases de equivalencia son disjuntas por pares, es decir, si dos clases de equivalencia  $\{x \in A \mid aRx\}$  y  $\{y \in A \mid bRy\}$  tienen un elemento  $c$  en común, entonces son de hecho el mismo conjunto. Supongamos que  $aRc$  y  $bRc$ . Por simetría,  $cRb$ , y por transitividad,  $aRb$ . Así, para cualquier elemento arbitrario  $x \in \{x \in A \mid aRx\}$ , tenemos  $xRa$  y, por transitividad,  $xRb$ , y por tanto  $x \in \{x \in A \mid bRy\}$ , y por tanto  $\{x \in A \mid aRx\} = \{x \in A \mid bRy\}$ .

Ob. Del mismo modo,  $\{x \in A \mid bRy\} = \{x \in A \mid cRx\}$

Para la segunda parte de la prueba, sea  $\{A_i\}_{i \in I}$  una partición de  $A$ , y defina  $R$  como  $R = \{(a, b) \in A^2 \mid \exists i \in I \text{ tal que } a \in A_i \text{ y } b \in A_i\}$ . Decimos que  $R$  es una relación de equivalencia sobre  $A$ . La reflexividad se cumple, ya que  $a \in A_i$  implica  $aRa$ . La simetría se cumple, porque si  $aRb$ , entonces  $a$  y  $b$  están en el mismo conjunto  $A_i$  y, por lo tanto,  $bRa$ .

Si  $aRb$  y  $bRc$ , entonces los tres elementos están en el mismo conjunto  $A_i$  y, por lo tanto, se mantienen  $aRc$  y la transitividad. Para ver que los conjuntos en la partición son las clases de equivalencia de  $R$ , observe que si  $a \in A_i$ , entonces  $x \in \{x \in A \mid xRa\}$  implica  $x \in A_i$ , y  $x \in A_i$  implica  $xRa$ .

■ Una relación binaria  $R$  en un conjunto  $A$  es antisimétrica si

$aRb$  y  $bRa$  implican  $a = b$

Por ejemplo, la relación “” sobre los números naturales es antisimétrica, ya que  $aRb$  y  $bRa$  implican  $a = b$ . Una relación que es reflexiva, antisimétrica y transitiva es un orden parcial, y llamamos conjunto parcialmente ordenado a un conjunto en el que se define un orden parcial. Por ejemplo, la relación “es descendiente de” es un orden parcial en el conjunto de todas las personas (si consideramos a los individuos como sus propios descendientes).

En un conjunto  $A$  parcialmente ordenado, puede no haber un solo elemento “máximo” a tal que  $bRa$  para todo  $b \in A$ . En cambio, el conjunto puede contener varios elementos máximos a tales que para ningún  $b \in A$ , donde  $b \neq a$ , es el caso que  $aRb$ . Por ejemplo, una colección de cajas de diferentes tamaños puede contener varias cajas máximas que no caben dentro de ninguna otra caja, pero no tiene una sola caja “máxima” en la que quepa

cualquier otra caja.<sup>3</sup> Una relación  $R$  en un conjunto  $A$  es una relación total si para todo  $a; b \in A$ , tenemos  $aRb$  o  $bRa$  (o ambos), es decir, si todo par de elementos de  $A$  está relacionado por  $R$ . Un orden parcial que es también una relación total es un orden total u orden lineal . Por ejemplo, la relación “” es un orden total sobre los números naturales, pero la relación “es descendiente de” no es un orden total sobre el conjunto de todas las personas, ya que hay individuos que ninguno desciende del otro. Una relación total que es transitiva, pero no necesariamente reflexiva y antisimétrica, es un preorden total.

## Ejercicios

### B.2-1

Demuestre que la relación de subconjuntos “” en todos los subconjuntos de  $Z$  es un orden parcial pero no un orden total.

### B.2-2

Demostrar que para cualquier entero positivo  $n$ , la relación “módulo equivalente  $n$ ” es un relación de alencia sobre los enteros. (Decimos que  $a \equiv b \pmod{n}$  si existe una un número entero  $q$  tal que  $a - b = qn$ .) ¿En qué clases de equivalencia divide esta relación los números enteros?

### B.2-3

Dar ejemplos de relaciones que son

- reflexivo y simétrico pero no transitivo,
- reflexivo y transitivo pero no simétrico,
- simétrica y transitiva pero no reflexiva.

---

<sup>3</sup>Para ser precisos, para que la relación de “encajar dentro” sea un orden parcial, necesitamos ver una caja como encajando dentro de sí misma.

## B.2-4

Sea  $S$  un conjunto finito y sea  $R$  una relación de equivalencia sobre  $S$ . Demuestre que si además  $R$  es antisimétrico, entonces las clases de equivalencia de  $S$  con respecto a  $R$  son singletons.

## B.2-5

El profesor Narciso afirma que si una relación  $R$  es simétrica y transitiva, entonces también es reflexiva. Ofrece la siguiente prueba. Por simetría,  $aRb$  implica  $bRa$ .

La transitividad, por lo tanto, implica  $aRa$ . ¿Está en lo correcto el profesor?

### B.3 Funciones

Dados dos conjuntos  $A$  y  $B$ , una función  $f$  es una relación binaria sobre  $A$  y  $B$  tal que para todo  $a \in A$ , existe precisamente un  $b \in B$  tal que  $.a; b \in f$ . El conjunto  $A$  se llama dominio de  $f$ , y el conjunto  $B$  se llama codominio de  $f$ . A veces escribimos  $f : A \rightarrow B$ ; y si  $.a; b \in f$ , escribimos  $b = f(a)$ , ya que  $b$  está únicamente determinado por la elección de  $a$ .

Intuitivamente, la función  $f$  asigna un elemento de  $B$  a cada elemento de  $A$ . A ningún elemento de  $A$  se le asignan dos elementos diferentes de  $B$ , pero el mismo elemento de  $B$  se le puede asignar a dos elementos diferentes de  $A$ . Por ejemplo, la relación binaria

$f : \{0, 1\} \rightarrow \{0, 1, 2\}$  dada por  $f(0) = 1$  y  $f(1) = 2$

es una función  $f : \mathbb{N} \rightarrow \mathbb{N}$ , ya que para cada número natural  $a$ , hay exactamente un valor  $b$  en  $\{0, 1, 2\}$  tal que  $b = f(a)$ . Para este ejemplo,  $f(0) = 1$ ,  $f(1) = 2$ , etc. En contraste, la relación binaria

$g : \{0, 1\} \rightarrow \{0, 1, 2\}$  dada por  $g(0) = 0$  y  $g(1) = 0$

no es una función, ya que  $.0; 1 \in g$  y  $.1; 0 \in g$ , y así para la elección  $a = 1$ , no hay precisamente un  $b$  tal que  $.a; b \in g$ .

Dada una función  $f : A \rightarrow B$ , si  $b \in f(a)$ , decimos que  $a$  es el argumento de  $f$  y que  $b$  es el valor de  $f$  en  $a$ . Podemos definir una función indicando su valor para cada elemento de su dominio. Por ejemplo, podríamos definir  $f : \mathbb{N} \rightarrow \mathbb{N}$  para  $n \in \mathbb{N}$ , lo que significa  $f(n) = n^2$ . Dos funciones  $f$  y  $g$  son iguales si tienen el mismo dominio y codominio y si, para todo  $a$  en el dominio,  $f(a) = g(a)$ .

Una sucesión finita de longitud  $n$  es una función  $f$  cuyo dominio es el conjunto de  $n$  enteros  $\{0, 1, \dots, n-1\}$ . A menudo denotamos una secuencia finita listando sus valores:  $f(0), f(1), \dots, f(n-1)$ . Una sucesión infinita es una función cuyo dominio es el conjunto  $\mathbb{N}$  de números naturales. Por ejemplo, la sucesión de Fibonacci, definida por la recurrencia (3.22), es la sucesión infinita  $f(0) = 1, f(1) = 1, f(2) = 2, f(3) = 3, f(4) = 5, f(5) = 8, f(6) = 13, f(7) = 21, \dots$ .

Cuando el dominio de una función  $f$  es un producto cartesiano, a menudo omitimos los paréntesis adicionales que rodean el argumento de  $f$ . Por ejemplo, si tuviéramos una función  $f : A_1 A_2 \dots A_n \rightarrow B$ , escribiríamos  $b \in f(a_1; a_2; \dots; a_n)$  en lugar de  $b \in f(a_1; a_2; \dots; a_n)$ . También llamamos a cada  $a_i$  un argumento de la función  $f$ , aunque técnicamente el (único) argumento de  $f$  es la  $n$ -tupla  $(a_1; a_2; \dots; a_n)$ .

Si  $f : A \rightarrow B$  es una función y  $b \in f(a)$ , entonces decimos que  $b$  es la imagen de  $a$  bajo  $f$ . La imagen de un conjunto  $A_0 \subseteq A$  bajo  $f$  está definida por  $f(A_0) = \{f(a) \mid a \in A_0\}$ .

$B = f(A)$  para algún  $A \subseteq A_0$ : El rango de  $f$  es la imagen

de su dominio, es decir,  $f(A)$ . Por ejemplo, el rango de la función  $f : \mathbb{N} \rightarrow \mathbb{N}$  definido por  $f(n) = 2n$  es  $\mathbb{N}$ . De hecho,  $f(\mathbb{N}) = \{2n \mid n \in \mathbb{N}\}$  para algún  $n \in \mathbb{N}$ , en otras palabras, el conjunto de enteros pares no negativos.

Una función es sobreyectiva si su recorrido es su codominio. Por ejemplo, la función  $f : \mathbb{N} \rightarrow \mathbb{N}$  definida por  $f(n) = 2n$  es una función sobreyectiva de  $\mathbb{N}$  a  $\mathbb{N}$ , ya que cada elemento en  $\mathbb{N}$  aparece como el valor de  $f$  para algún argumento. Por el contrario, la función  $f : \mathbb{N} \rightarrow \mathbb{N}$  definida por  $f(n) = n^2$  no es una función sobreyectiva de  $\mathbb{N}$  a  $\mathbb{N}$ , ya que ningún argumento de  $f$  puede producir 3 como valor. La función  $f : \mathbb{N} \rightarrow \mathbb{N}$  es, sin embargo, una función sobreyectiva de los números naturales a los números pares. Una sobreyección  $f : A \rightarrow B$  a veces se describe como la aplicación de  $A$  en  $B$ . Cuando decimos que  $f$  es sobre, queremos decir que es sobreyectiva.

Una función  $f : A \rightarrow B$  es una inyección si argumentos distintos de  $f$  producen valores distintos, es decir, si  $a \neq a'$  implica  $f(a) \neq f(a')$ . Por ejemplo, la función  $f : \mathbb{N} \rightarrow \mathbb{N}$  definida por  $f(n) = 2n$  es una función inyectiva de  $\mathbb{N}$  a  $\mathbb{N}$ , ya que cada número par  $b$  es la imagen bajo  $f$  de a lo sumo un elemento del dominio, a saber,  $b=2$ . La función  $f : \mathbb{N} \rightarrow \mathbb{N}$  definida por  $f(n) = n^2$  no es inyectiva, ya que el valor 1 es producido por dos argumentos: 2 y 3. Una inyección a veces se denomina función uno a uno.

Una función  $f : A \rightarrow B$  es biyectiva si es sobreyectiva e inyectiva. Por ejemplo, la función  $f : \mathbb{N} \rightarrow \mathbb{N}$  definida por  $f(n) = 2n$  es una biyección de  $\mathbb{N}$  a  $\mathbb{N}$ .

$$\begin{array}{rcl} & \vdots & \\ 1 & 1 & 2 & 1 & \vdots \\ & \vdots & \\ 3 & 2 & 4 & 1 & 2 & \vdots \\ & \vdots & \\ & \vdots & \end{array}$$

La función es inyectiva, ya que ningún elemento de  $\mathbb{N}$  es imagen de más de un elemento de  $\mathbb{N}$ . Es sobreyectiva, ya que todo elemento de  $\mathbb{N}$  aparece como imagen de algún elemento de  $\mathbb{N}$ . Por lo tanto, la función es biyectiva. Una biyección a veces se denomina correspondencia uno a uno, ya que empareja elementos en el dominio y el codominio. Una biyección de un conjunto  $A$  consigo mismo a veces se denomina permutación.

Cuando una función  $f$  es biyectiva, definimos su inversa  $f^{-1}$  como

$f^{-1}(b) = a$  si y solo si  $f(a) = b$

Por ejemplo, la inversa de la función  $f : \mathbb{N} / D \rightarrow \mathbb{N}$  es

$$\begin{aligned} & \text{si } m \geq 0 \\ f^{-1}(m) / D &= \begin{cases} 2m & \text{si } m \geq 0 \\ 2m+1 & \text{si } m < 0 \end{cases} \end{aligned}$$

### Ejercicios

#### B.3-1

Sean A y B conjuntos finitos, y sea  $f : A \rightarrow B$  una función. Muestra que

- a. si  $f$  es inyectiva, entonces  $|A| \leq |B|$ ;
- b. si  $f$  es sobreyectiva, entonces  $|A| \geq |B|$ .

#### B.3-2

¿La función  $f : \mathbb{N} / D \rightarrow \mathbb{N}$  es biyectiva cuando el dominio y el codominio son  $\mathbb{N}$ ?

¿Es biyectiva cuando el dominio y el codominio son  $\mathbb{Z}$ ?

#### B.3-3

Dé una definición natural para el inverso de una relación binaria tal que si una relación es de hecho una función biyectiva, su inverso relacional es su inverso funcional.

#### B.3-4 ?

Dar una biyección de  $\mathbb{Z}$  a  $\mathbb{Z}^2$ .

## B.4 Gráficos

Esta sección presenta dos tipos de grafos: dirigidos y no dirigidos. Ciertas definiciones en la literatura difieren de las dadas aquí, pero en su mayor parte, las diferencias son leves. La sección 22.1 muestra cómo podemos representar gráficas en la memoria de la computadora.

Un grafo dirigido (o digrafo)  $G$  es un par  $(V; E)$ , donde  $V$  es un conjunto finito y  $E$  es una relación binaria sobre  $V$ . El conjunto  $V$  se llama conjunto de vértices de  $G$ , y sus elementos se llaman vértices (singular: vértice). El conjunto  $E$  se llama conjunto de aristas de  $G$ , y sus elementos se llaman aristas. La figura B.2(a) es una representación pictórica de un grafo dirigido sobre el conjunto de vértices  $\{1; 2; 3; 4; 5; 6\}$ . Los vértices están representados por círculos en la figura y los bordes están representados por flechas. Tenga en cuenta que los bucles automáticos (bordes desde un vértice hacia sí mismo) son posibles.

En un grafo no dirigido  $(V; E)$ , el conjunto de aristas  $E$  consta de pares desordenados de vértices, en lugar de pares ordenados. Es decir, una arista es un conjunto  $\{u; v\}$ , donde

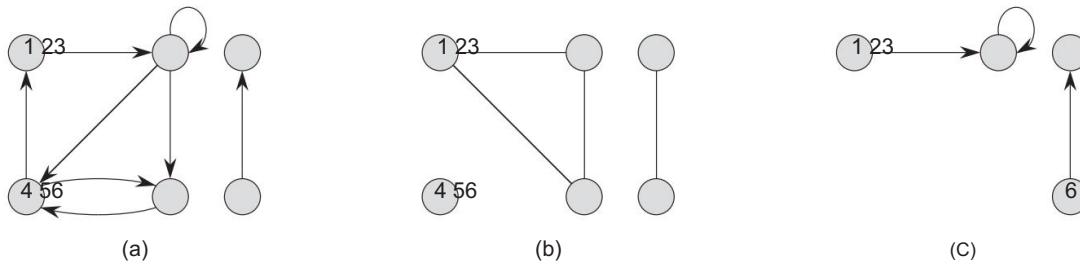


Figura B.2 Grafos dirigidos y no dirigidos. (a) Un grafo dirigido  $G.D .V; E/$ , donde  $VD f1; 2; 3; 4; 5; 6g$  y  $DE f.1; 2/; .2; 2/; .2; 4/; .2; 5/; .4; 1/; .4; 5/; .5; 4/; .6; 3/g$ . El borde  $.2; 2/$  es un bucle automático. (b) Un grafo no dirigido  $G.D .V; E/$ , donde  $VD f1; 2; 3; 4; 5; 6g$  y  $DE f.1; 2/; .1; 5/; .2; 5/; .3; 6/g$ . El vértice 4 está aislado. (c) El subgrafo del grafo del inciso (a) inducido por el conjunto de vértices  $f1; 2; 3; 6 g$ .

tu; 2 V y tu  $\not\propto$ . Por convención, usamos la notación  $.u; /$  para una arista, en lugar de la notación de conjunto  $f u; g$ , y consideramos  $.u; /$  y  $; u/$  para ser el mismo borde.

En un gráfico no dirigido, los bucles automáticos están prohibidos, por lo que cada borde consta de dos vértices distintos. La Figura B.2(b) es una representación pictórica de un grafo no dirigido en el conjunto de vértices  $f1; 2; 3; 4; 5; 6 g$ .

Muchas definiciones de grafos dirigidos y no dirigidos son las mismas, aunque ciertos términos tienen significados ligeramente diferentes en los dos contextos. Si  $tu; /$  es una arista en un grafo dirigido  $G.D .V; E/$ , decimos que  $.u; /$  es incidente o sale del vértice  $u$  y es incidente o entra en el vértice  $t$ . Por ejemplo, los bordes que salen del vértice 2 en la Figura B.2(a) son  $.2; 2/; .2; 4/$ , y  $.2; 5/$ . Las aristas que entran al vértice 2 son  $.1; 2/$  y  $.2; 2/$ . Si  $tu; /$  es una arista en un grafo no dirigido  $G.D .V; E/$ , decimos que  $.u; /$  es incidente en los vértices  $u$  y  $t$ . En la figura B.2(b), las aristas incidentes en el vértice 2 son  $.1; 2/$  y  $.2; 5/$ .

Si  $tu; /$  es una arista en un grafo  $G.D .V; E/$ , decimos que el vértice  $t$  es adyacente al vértice  $u$ . Cuando el gráfico no está dirigido, la relación de adyacencia es simétrica. Cuando el gráfico es dirigido, la relación de adyacencia no es necesariamente simétrica. Si es adyacente a  $u$  en un gráfico dirigido, a veces escribimos  $u ! t$ . En las partes (a) y (b) de la figura B.2, el vértice 2 es adyacente al vértice 1, ya que la arista  $.1; 2/$  pertenece a ambas gráficas. El vértice 1 no es adyacente al vértice 2 en la figura B.2(a), ya que la arista  $.2; 1/$  no pertenece a la gráfica.

El grado de un vértice en un gráfico no dirigido es el número de aristas que inciden sobre él. Por ejemplo, el vértice 2 en la Figura B.2(b) tiene grado 2. Un vértice cuyo grado es 0, como el vértice 4 en la Figura B.2(b), está aislado. En un gráfico dirigido, el grado de salida de un vértice es el número de aristas que salen de él, y el grado de entrada de un vértice es el número de aristas que entran en él. El grado de un vértice en un gráfico dirigido es su in-

grado más su grado de salida. El vértice 2 en la Figura B.2(a) tiene un grado de entrada 2, un grado de salida 3 y un grado 5.

Un camino de longitud  $k$  desde un vértice  $u$  hasta un vértice  $u_0$  en un grafo  $G = (V, E)$  es una secuencia  $h_0; 1; 2; \dots; k$  de vértices tales que  $u \in D_{h_i}$  para  $i \in D_{h_i}$ ;  $2; \dots; k$ . La longitud del camino es el número de aristas en el camino. El camino contiene los vértices y las aristas  $h_0; 1; 2; \dots; k$ . (Siempre hay un camino de longitud 0 de  $u$  a  $u$ .) Si hay un camino  $p$  de  $u$  a  $u_0$ , decimos que  $u_0$  es accesible desde  $u$  a través de  $p$ , lo que significa,  $u_0$  si  $G$  está dirigido. Un camino es simple<sup>4</sup> si todos los vértices en el camino a veces escrito como  $u$  son distintos. En la Figura B.2(a), el camino  $h_1; 2; 5; 4; i_1$  es un camino simple de longitud 3. El camino  $h_2; 2; 5; 4; i_1$  no es simple.

Un subtrayecto del trayecto  $p$  de  $h_0; 1; \dots; k$  es una subsecuencia contigua de sus vértices. Es decir, para cualquier  $0 \leq i < j \leq k$ , la subsecuencia de vértices  $h_i; h_{i+1}; \dots; h_j$  es un subcamino de  $p$ .

En un grafo dirigido, un camino  $h_0; 1; \dots; k$  forma un ciclo si  $h_0 = h_k$  y el camino contiene al menos un borde. El ciclo es simple si, además,  $h_1; 2; \dots; k$  son distintos. Un bucle automático es un ciclo de longitud 1. Dos caminos  $h_0; 1; 2; \dots; k_1; 0$  y  $h_0; 0; 1; \dots; k_2; 0$  forman el mismo ciclo si existe un entero  $j$  tal que  $k_1 = j$  y  $k_2 = j + 1$ . En la figura B.2(a), el camino  $h_1; 2; 4; 1; i_1$  forma el mismo ciclo que  $h_1; 2; 4; 1; i_1$ . Un grafo dirigido sin bucles automáticos es simple. En un grafo no dirigido, un camino  $h_0; 1; \dots; k$  forma un ciclo si  $k \geq 3$  y  $D_k$ ; el ciclo es simple si  $h_1; 2; \dots; k$  son distintos. Por ejemplo, en la figura B.2(b), el camino  $h_1; 2; 5; 1; i_1$  es un ciclo simple. Un grafo sin ciclos es acíclico.

Un grafo no dirigido es conexo si todos los vértices son accesibles desde todos los demás vértices. Los componentes conectados de un grafo son las clases de equivalencia de vértices bajo la relación "es accesible desde". El gráfico de la figura B.2(b) tiene tres componentes conectados:  $f_1; 2; 5g$ ,  $f_3; 6g$  y  $f_4g$ . Cada vértice en  $f_1; 2; 5g$  es accesible desde cualquier otro vértice en  $f_1; 2; 5g$ . Un grafo no dirigido es conexo si tiene exactamente una componente conexa. Las aristas de una componente conexa son aquellas que inciden únicamente en los vértices de la componente; en otras palabras, borde  $u; v$  es una arista de un componente conexo solo si  $u$  y  $v$  son vértices del componente.

Un grafo dirigido está fuertemente conectado si cada dos vértices son accesibles entre sí. Las componentes fuertemente conexas de un grafo dirigido son las equivalentes

---

<sup>4</sup>Algunos autores se refieren a lo que llamamos un camino como un "paseo" y a lo que llamamos un camino simple como simplemente un "camino". Usamos los términos "camino" y "camino simple" a lo largo de este libro de manera consistente con sus definiciones.

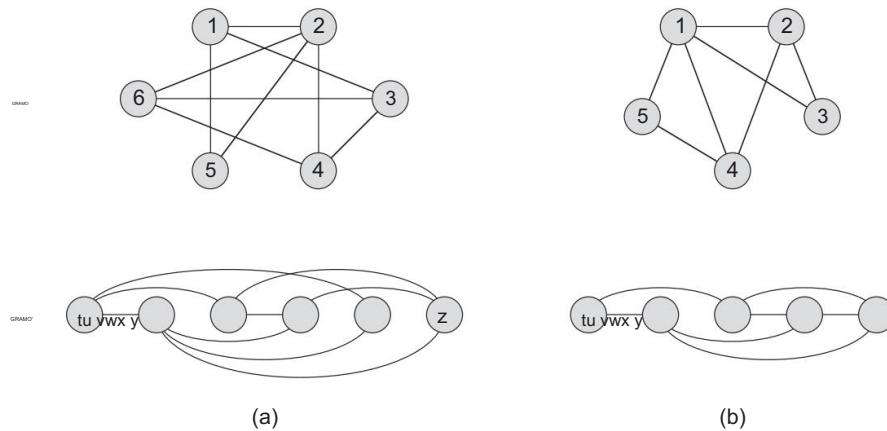


Figura B.3 (a) Un par de gráficas isomorfas. Los vértices del gráfico superior se asignan a los vértices del gráfico inferior mediante  $f: V_1 \rightarrow V_2$ ;  $f(1) = D$ ;  $f(2) = D'$ ;  $f(3) = Dw$ ;  $f(4) = Dx$ ;  $f(5) = Dy$ ;  $f(6) = D'$ . (b) Dos gráficas que no son isomorfas, ya que la gráfica superior tiene un vértice de grado 4 y la gráfica inferior no.

alence clases de vértices bajo la relación "son mutuamente alcanzables". Un grafo dirigido es fuertemente conexo si solo tiene un componente fuertemente conexo. El gráfico de la figura B.2(a) tiene tres componentes fuertemente conectados:  $f_1$ ;  $2$ ;  $4$ ;  $5g$ ,  $f_3g$  y  $f_6g$ . Todos los pares de vértices en  $f_1$ ;  $2$ ;  $4$ ;  $5g$  son mutuamente accesibles. Los vértices  $f_3$ ;  $6g$  no forman un componente fuertemente conexo, ya que el vértice  $6$  no se puede alcanzar desde el vértice  $3$ .

Dos gráficos  $G$  y  $G'$  son isomorfos si existe un mapeo  $f: V(G) \rightarrow V(G')$  tal que  $f(u)$  es vecino de  $f(v)$  en  $G'$  si y sólo si  $u$  es vecino de  $v$  en  $G$ . En otras palabras, podemos volver a etiquetar los vértices de  $G$  para que sean vértices de  $G'$  manteniendo las aristas correspondientes en  $G$  y  $G'$ . La figura B.3(a) muestra un par de gráficos isomórficos  $G$  y  $G'$  con conjuntos de vértices respectivos  $V(G) = \{f_1, f_2, f_3, f_4, f_5, f_6\}$  y  $V(G') = \{D_x, D_w, D_y, D_z, D_u, D_v\}$ . El mapeo de  $V(G)$  a  $V(G')$  dado por  $f(f_1) = D_x, f(f_2) = D_w, f(f_3) = D_y, f(f_4) = D_z, f(f_5) = D_u$  y  $f(f_6) = D_v$  proporciona la función biyectiva requerida. Los gráficos de la figura B.3(b) no son isomorfos. Aunque ambos gráficos tienen 5 vértices y 7 aristas, el gráfico superior tiene un vértice de grado 4 y el gráfico inferior no.

Decimos que un grafo  $G_0$  D.  $V^0 : E_0$  / es un subgrafo de  $G$  D.  $V$ ; si  $V^0 \subseteq V$  y  $E_0 \subseteq E$ . Dado un conjunto  $V^0 \subseteq V$ , subgrafo de  $G$  inducido por  $V^0$  es el gráfico  $G_0$  D.  $V^0 : E_0$  /, donde

E0 Dfu: / 2 EW u: 2 voltios

El subgrafo inducido por el conjunto de vértices f1; 2; 3; 6g en la Figura B.2(a) aparece en la Figura B.2(c) y tiene el conjunto de bordes f.1; 2/; .2; 2/; .6; 3/g.

Dado un grafo no dirigido  $G = (V, E)$ , la versión dirigida de  $G$  es el grafo dirigido  $G_0 = (V, E_0)$ , donde  $(u, v) \in E_0$  si y solo si  $(v, u) \in E$ . Es decir, reemplazamos cada arista no dirigida  $(u, v)$  en  $G$  por las dos aristas dirigidas  $(u, v)$  y  $(v, u)$  en la versión dirigida. Dado un grafo dirigido  $G = (V, E)$ , la versión no dirigida de  $G$  es el grafo no dirigido  $G_0 = (V, E_0)$ , donde  $(u, v) \in E_0$  si y sólo si  $(u, v) \in E$ . Es decir, la versión no dirigida contiene los bordes de  $G$  "con sus direcciones eliminadas" y con los bucles propios eliminados. (Dado que  $(u, v) \in E$  son la misma arista en un grafo no dirigido, la versión no dirigida de un grafo dirigido lo contiene solo una vez, incluso si el grafo dirigido contiene ambas aristas  $(u, v)$  y  $(v, u)$ .) En un grafo dirigido  $G = (V, E)$ , vecino de un vértice  $u$  es cualquier vértice adyacente a  $u$  en la versión no dirigida de  $G$ . Es decir, es vecino de  $u$  si  $(u, v) \in E$  o  $(v, u) \in E$ . En un grafo no dirigido,  $u$  y  $v$  son vecinos si son adyacentes.

Varios tipos de gráficos tienen nombres especiales. Un grafo completo es un grafo no dirigido en el que todos los pares de vértices son adyacentes. Un grafo bipartito es un grafo no dirigido  $G = (V, E)$  en la que  $V$  puede dividirse en dos conjuntos  $V_1$  y  $V_2$  tales que  $(u, v) \in E$  implica  $u \in V_1$  y  $v \in V_2$  o  $u \in V_2$  y  $v \in V_1$ . Es decir, todas las aristas van entre los dos conjuntos  $V_1$  y  $V_2$ . Un grafo acíclico no dirigido es un bosque, y un grafo acíclico no dirigido conexo es un árbol (libre) (ver Sección B.5). A menudo tomamos las primeras letras de "gráfico acíclico dirigido" y llamamos dag a dicho gráfico.

Hay dos variantes de gráficos que puede encontrar ocasionalmente. Un gráfico múltiple es como un gráfico no dirigido, pero puede tener múltiples bordes entre vértices y bucles propios. Un hipergráfico es como un gráfico no dirigido, pero cada hiperarista, en lugar de conectar dos vértices, conecta un subconjunto arbitrario de vértices. Muchos algoritmos escritos para gráficos ordinarios dirigidos y no dirigidos se pueden adaptar para ejecutarse en estas estructuras similares a gráficos.

La contracción de un grafo no dirigido  $G = (V, E)$  por una arista  $e = (u, v)$  es un grafo  $G_0 = (V, E_0)$ , donde  $\{u, v\} \setminus e \cup \{w\}$  es un nuevo vértice. El conjunto de aristas  $E_0$  se forma a partir de  $E$  eliminando la arista  $(u, v)$ , para cada vértice  $w$ , borrando cualquiera de  $(u, w)$  y  $(v, w)$ ; efecto,  $u$  y  $v$  están en  $E_0$  y agregando el incidente en  $w$  o nueva arista  $(w, x)$ . En "contraídos" en un solo vértice.

## Ejercicios

### B.4-1

Los asistentes a una fiesta de profesores se dan la mano para saludarse, y cada profesor recuerda cuántas veces se dio la mano. Al final de la fiesta, el jefe de departamento suma el número de veces que cada profesor se dio la mano.

Muestre que el resultado es par probando el lema del apretón de manos: si  $G = \langle V, E \rangle$  es un grafo no dirigido, entonces

X grado./ D 2  $\sum_{j \in E} j$  :

$\sum_{j \in E}$

#### B.4-2

Muestre que si un grafo dirigido o no dirigido contiene un camino entre dos vértices  $u$  y  $v$ , luego contiene un camino simple entre  $u$  y  $v$ . Demuestre que si un gráfico dirigido contiene un ciclo, entonces contiene un ciclo simple.

#### B.4-3

Muestre que cualquier grafo conexo no dirigido  $G = \langle V, E \rangle$  satisface  $\sum_{j \in E} j \leq |V| - 1$ .

#### B.4-4

Verifique que en un grafo no dirigido, la relación "es accesible desde" es una relación de equivalencia en los vértices del grafo. ¿Cuál de las tres propiedades de una relación de equivalencia se cumple en general para la relación "es accesible desde" en los vértices de un gráfico dirigido?

#### B.4-5

¿Cuál es la versión no dirigida del grafo dirigido de la figura B.2(a)? ¿Cuál es la versión dirigida del grafo no dirigido de la figura B.2(b)?

#### B.4-6 ?

Demuestre que podemos representar una hipergrafía mediante una gráfica bipartita si hacemos que la incidencia en la hipergrafía corresponda a la adyacencia en la gráfica bipartita. (Sugerencia: deje que un conjunto de vértices en el gráfico bipartito corresponda a los vértices del hipergráfico y deje que el otro conjunto de vértices del gráfico bipartito corresponda a los hiperbordes).

## B.5 Árboles

Al igual que con los gráficos, hay muchas nociones de árboles relacionadas, pero ligeramente diferentes. Esta sección presenta definiciones y propiedades matemáticas de varios tipos de árboles. Las secciones 10.4 y 22.1 describen cómo podemos representar árboles en la memoria de la computadora.

### B.5.1 Árboles libres

Como se define en la Sección B.4, un árbol libre es un grafo no dirigido, acíclico y conexo. A menudo omitimos el adjetivo "libre" cuando decimos que un gráfico es un árbol. Si un gráfico no dirigido es acíclico pero posiblemente desconectado, es un bosque. Muchos algoritmos que funcionan

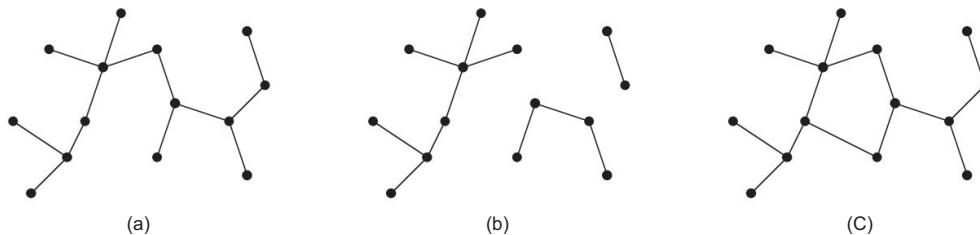


Figura B.4 (a) Un árbol libre. (b) Un bosque. (c) Un grafo que contiene un ciclo y por lo tanto no es ni un árbol ni un bosque.

para los árboles también funcionan para los bosques. La Figura B.4(a) muestra un árbol libre y la Figura B.4(b) muestra un bosque. El bosque de la Figura B.4(b) no es un árbol porque no está conectado. El gráfico de la figura B.4(c) es conexo, pero no es un árbol ni un bosque, porque contiene un ciclo.

El siguiente teorema captura muchos hechos importantes acerca de los árboles libres.

#### Teorema B.2 (Propiedades de los árboles libres)

Sea  $G = \langle V; E \rangle$  sea un grafo no dirigido. Las siguientes declaraciones son equivalentes.

1.  $G$  es un árbol libre.
2. Dos vértices cualesquiera de  $G$  están conectados por un único camino simple.
3.  $G$  es conexo, pero si se elimina cualquier arista de  $E$ , el gráfico resultante se distorsiona conectado.
4.  $G$  es conexo y  $|E| = |V| - 1$ .
5.  $G$  es acíclico.
6.  $G$  es acíclico, pero si se agrega cualquier borde a  $E$ , el gráfico resultante contiene un ciclo.

Prueba (1) ) (2): Dado que un árbol es conexo, dos vértices cualesquiera en  $G$  están conectados por al menos un camino simple. Suponga, en aras de la contradicción, que los vértices  $u$  y  $v$  están conectados por dos trayectorias simples  $p_1$  y  $p_2$ , como se muestra en la figura B.5.

Sea  $w$  el vértice en el que los caminos divergen por primera vez; es decir,  $w$  es el primer vértice tanto en  $p_1$  como en  $p_2$  cuyo sucesor en  $p_1$  es  $x$  y cuyo sucesor en  $p_2$  es  $y$ , donde  $x \neq y$ . Sea  $'$  el primer vértice en el que los caminos vuelven a converger; es decir,  $'$  es el primer vértice que sigue a  $w$  en  $p_1$  que también está en  $p_2$ . Sea  $p_0$  el subcamino de  $p_1$  desde  $w$  a través de  $x$  hasta  $'$ , y sea  $p_{00}$  el subcamino de  $p_2$  desde  $w$  a través de  $y$  hasta  $'$ .

Los caminos  $p_0$  y  $p_{00}$  no comparten vértices excepto sus puntos finales. Por lo tanto, el camino obtenido al concatenar  $p_0$  y el reverso de  $p_{00}$  es un ciclo, lo que contradice nuestra suposición

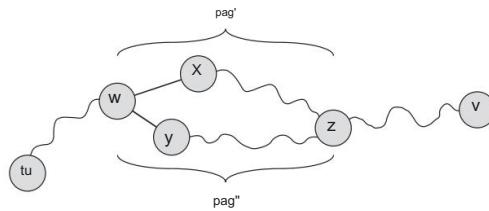


Figura B.5 Un paso en la demostración del teorema B.2: si (1)  $G$  es un árbol libre, entonces (2) dos vértices cualesquiera de  $G$  están conectados por un único camino simple. Supongamos por el bien de la contradicción que los vértices  $u$  y  $v$  están conectados por dos caminos simples distintos  $p_1$  y  $p_2$ . Estos caminos divergen primero en el vértice  $w$  y vuelven a converger en el vértice  $'$ . El camino  $p_0$  concatenado con el reverso del camino  $p_0$  forma un ciclo, lo que produce la contradicción.

que  $G$  es un árbol. Así, si  $G$  es un árbol, puede haber a lo sumo un camino simple entre dos vértices.

(2) ) (3): Si dos vértices cualesquiera en  $G$  están conectados por un único camino simple, entonces  $G$  es conexo. Dejarte; / sea cualquier arista en  $E$ . Esta arista es un camino desde  $u$ , hasta  $y$  y por lo tanto debe ser el único camino desde  $u$  hasta  $y$ . Si eliminamos .u; / desde  $G$ , no hay camino desde  $u$  hasta  $y$ , por lo tanto, su eliminación desconecta  $G$ .

(3) ) (4): Por suposición, el gráfico  $G$  es conexo y, por el ejercicio B.4-3, tenemos jEj 1. Probaremos jEjV j 1 por inducción. Un grafo conexo con  $n \leq 1$  o  $n = 2$  vértices tiene  $n - 1$  aristas. Suponga que  $G$  tiene  $n \geq 3$  vértices y que todas las gráficas que satisfacen (3) con menos de  $n$  vértices también satisfacen jEjV j 1. Quitar una arista arbitraria de  $G$  separa la gráfica en  $k \geq 2$  componentes conectados (en realidad  $k \geq 2$ ). Cada componente satisface (3), o de lo contrario  $G$  no cumpliría (3). Si consideramos cada componente conexo  $V_i$ , con conjunto de aristas  $E_i$ , como su propio árbol libre, entonces debido a que cada componente tiene menos de  $jV_j$  vértices, por la hipótesis inductiva tenemos jEi jVij 1. Por lo tanto, el número de aristas en todos los componentes combinado es como máximo  $jV_j k jV_j 2$ . Al sumar el borde eliminado se obtiene jEjV j 1. (4) ) (5): Suponga que  $G$  es conexo y que jEjD jVj que  $G$  es 1.

acíclico. Suponga que  $G$  tiene un ciclo que contiene  $k$  vértices y 1. Debemos mostrar sin pérdida de generalidad suponga que este ciclo es simple. Sea  $G_k \subseteq D$ .  $V_k = \{v_1, v_2, \dots, v_k\}$ ,  $E_k$  sea el subgrafo de  $G$  consistente en el ciclo. Tenga en cuenta que  $jV_k \leq jE_k \leq k$ .

Si  $k < jV_j$ , debe haber un vértice  $v_{k+1} \in V_k$  que sea adyacente a algún vértice  $v_i \in V_k$ , ya que  $G$  es conexo. Definir  $G_{k+1} \subseteq D$ .  $V_{k+1} = V_k \cup \{v_{k+1}\}$ ,  $E_{k+1}$  para ser el subgrafo de  $G$  con  $V_{k+1} \subseteq D$ .  $E_{k+1} = E_k \cup \{v_i v_{k+1}\}$ . Note que  $jV_{k+1} \leq jE_{k+1} \leq k + 1$ . Si  $k + 1 < jV_j$ , podemos continuar, definiendo  $G_{k+2}$  de la misma manera, y así sucesivamente, hasta obtener  $G_n \subseteq D$ .  $V_n = \{v_1, v_2, \dots, v_n\}$ , donde  $n \leq jV_j$ ,

$\forall v \in V \setminus V_n, \forall j \in E \setminus E_n, \forall v_j \in V \setminus V_n$ . Como  $G_n$  es un subgrafo de  $G$ , tenemos  $E_n \subseteq E$ . Entonces, por lo tanto  $j \in E \setminus E_n$ , lo que contradice la suposición de que  $E \setminus E_n$  es acíclico.

(5) ) (6): Supongamos que  $G$  es acíclico y que  $E \setminus E_n$  tiene número de componentes conexas de  $G$ . Cada componente conexa es un árbol libre por definición, y dado que (1) implica (5), la suma de todas bordes en todos los componentes conectados  $k$ . En componente de  $G$  es consecuencia, debemos tener  $k = 1$ , y  $G$  es de hecho un  $V \setminus V_n$  árbol. Dado que (1) implica (2), dos vértices cualesquiera en  $G$  están conectados por un único camino simple. Por lo tanto, agregar cualquier borde a  $G$  crea un ciclo.

(6) ) (1): Suponga que  $G$  es acíclico pero que agregar cualquier borde a  $E$  crea un ciclo. Debemos demostrar que  $G$  es conexo. Sean  $u$  y  $v$  vértices arbitrarios en  $G$ . Si  $u$  y  $v$  no son adyacentes, agregar el borde  $uv$  crea un ciclo en el que todos los bordes excepto  $uv$  pertenecen a  $G$ . Así, el ciclo menos arista  $uv$  debe contener un camino desde  $u$  hasta  $v$  como  $u$  y  $v$ , fueron elegidos arbitrariamente,  $G$  es conexo. ■

#### B.5.2 Árboles enraizados y ordenados

Un árbol enraizado es un árbol libre en el que uno de los vértices se distingue de los demás. Llamamos al vértice distinguido la raíz del árbol. A menudo nos referimos a un vértice de un árbol con raíz como un nodo<sup>5</sup> del árbol. La figura B.6(a) muestra un árbol con raíz en un conjunto de 12 nodos con raíz 7.

Considere un nodo  $x$  en un árbol con raíz  $T$  con raíz  $r$ . Llamamos a cualquier nodo  $y$  en el único camino simple de  $r$  a  $x$  un ancestro de  $x$ . Si  $y$  es un ancestro de  $x$ , entonces  $x$  es un descendiente de  $y$ . (Cada nodo es a la vez un ancestro y un descendiente de sí mismo.) Si  $y$  es un ancestro de  $x$  y  $x$  es un descendiente de  $y$ , entonces  $y$  es un ancestro propio de  $x$  y  $x$  es un descendiente propio de  $y$ . El subárbol con raíz en  $x$  es el árbol inducido por los descendientes de  $x$ , con raíz en  $x$ . Por ejemplo, el subárbol con raíz en el nodo 8 de la figura B.6(a) contiene los nodos 8, 6, 5 y 9.

Si la última arista del camino simple desde la raíz  $r$  de un árbol  $T$  hasta un nodo  $x$  es  $xy$ , entonces  $y$  es el padre de  $x$ , y  $x$  es un hijo de  $y$ . La raíz es el único nodo en  $T$  sin parentesco. Si dos nodos tienen el mismo parentesco, son hermanos. Un nodo sin hijos es una hoja o un nodo externo. Un nodo no hoja es un nodo interno.

---

<sup>5</sup>El término "nodo" se usa a menudo en la literatura de teoría de grafos como sinónimo de "vértice". Nos reservamos el término "nodo" significa un vértice de un árbol enraizado.

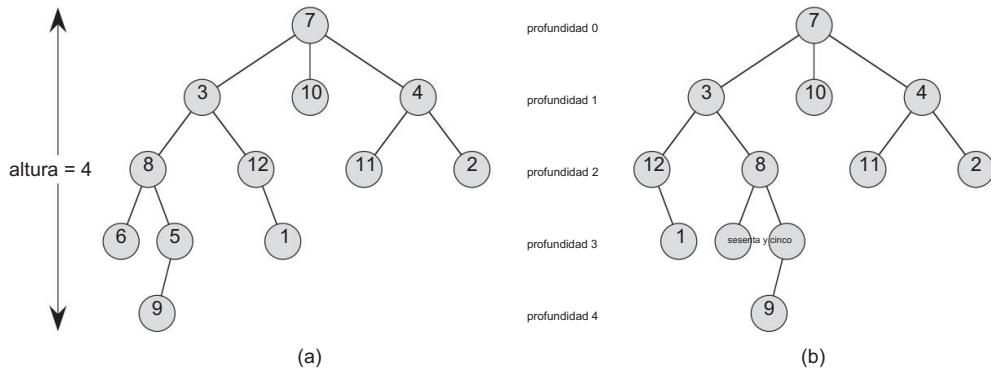


Figura B.6 Árboles enraizados y ordenados. (a) Un árbol enraizado con altura 4. El árbol se dibuja de manera estándar: la raíz (nodo 7) está en la parte superior, sus hijos (nodos con profundidad 1) están debajo, sus hijos (nodos con profundidad 2) están debajo de ellos, y así sucesivamente. Si el árbol está ordenado, importa el orden relativo de izquierda a derecha de los hijos de un nodo; de lo contrario no lo hace. (b) Otro árbol enraizado. Como árbol con raíz, es idéntico al árbol en (a), pero como árbol ordenado es diferente, ya que los hijos del nodo 3 aparecen en un orden diferente.

El número de hijos de un nodo  $x$  en un árbol con raíz  $T$  es igual al grado de  $x$ .  
La longitud del camino simple desde la raíz  $r$  hasta un nodo  $x$  es la profundidad de  $x$  en el nivel  $TA$  de un árbol que consta de todos los nodos a la misma profundidad. La altura de un nodo en un árbol es el número de aristas en el camino descendente simple más largo desde el nodo hasta una hoja, y la altura de un árbol es la altura de su raíz. La altura de un árbol también es igual a la mayor profundidad de cualquier nodo del árbol.

Un árbol ordenado es un árbol enraizado en el que se ordenan los hijos de cada nodo. Es decir, si un nodo tiene  $k$  hijos, entonces hay un primer hijo, un segundo hijo, . . . , y un  $k$ -ésimo hijo. Los dos árboles de la figura B.6 son diferentes cuando se consideran árboles ordenados, pero iguales cuando se consideran árboles con raíces.

### B.5.3 Árboles binarios y posicionales

Definimos árboles binarios recursivamente. Un árbol binario T es una estructura definida en un conjunto finito de nodos que

no contiene nodos o

6Observe que el grado de un nodo depende de si consideramos que T es un árbol con raíz o un árbol libre. El grado de un vértice en un árbol libre es, como en cualquier gráfico no dirigido, el número de vértices adyacentes. Sin embargo, en un árbol con raíz, el grado es el número de hijos: el padre de un nodo no cuenta para su grado.

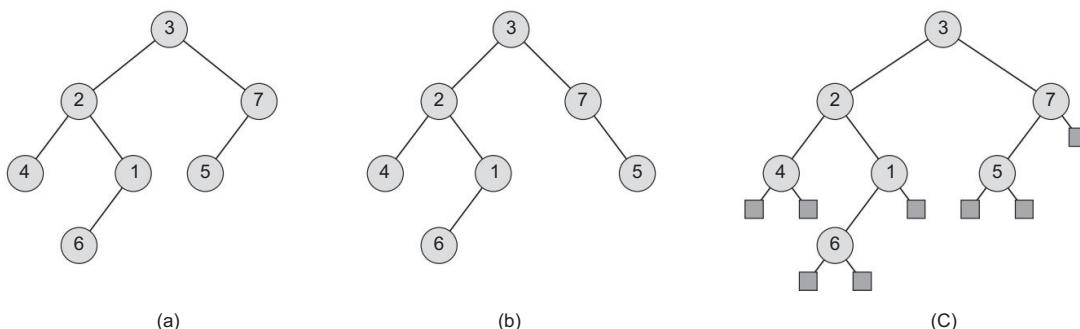


Figura B.7 Árboles binarios. (a) Un árbol binario dibujado de forma estándar. El hijo izquierdo de un nodo se dibuja debajo del nodo ya la izquierda. El niño derecho se dibuja debajo ya la derecha. (b) Un árbol binario diferente al de (a). En (a), el hijo izquierdo del nodo 7 es 5 y el hijo derecho está ausente. En (b), el hijo izquierdo del nodo 7 está ausente y el hijo derecho es 5. Como árboles ordenados, estos árboles son iguales, pero como árboles binarios, son distintos. (c) El árbol binario en (a) representado por los nodos internos de un árbol binario completo: un árbol ordenado en el que cada nodo interno tiene grado 2. Las hojas del árbol se muestran como cuadrados.

se compone de tres conjuntos disjuntos de nodos: un nodo raíz , un árbol binario llamado subárbol izquierdo y un árbol binario llamado subárbol derecho.

El árbol binario que no contiene nodos se denomina árbol vacío o árbol nulo, algunas veces denominado NIL. Si el subárbol izquierdo no está vacío, su raíz se llama hijo izquierdo de la raíz de todo el árbol. Del mismo modo, la raíz de un subárbol derecho no nulo es el hijo derecho de la raíz de todo el árbol. Si un subárbol es el árbol nulo NIL, decimos que el hijo está ausente o falta. La figura B.7(a) muestra un árbol binario.

Un árbol binario no es simplemente un árbol ordenado en el que cada nodo tiene un grado máximo de 2. Por ejemplo, en un árbol binario, si un nodo tiene solo un hijo, la posición del hijo, ya sea el hijo izquierdo o el derecho . niño—asuntos. En un árbol ordenado, no se distingue a un único hijo como izquierdo o derecho. La figura B.7(b) muestra un árbol binario que difiere del árbol de la figura B.7(a) debido a la posición de un nodo. Sin embargo, considerados como árboles ordenados, los dos árboles son idénticos.

Podemos representar la información de posicionamiento en un árbol binario mediante los nodos internos de un árbol ordenado, como se muestra en la figura B.7(c). La idea es reemplazar cada hijo que falta en el árbol binario con un nodo que no tenga hijos. Estos nodos de hoja se dibujan como cuadrados en la figura. El árbol resultante es un árbol binario completo: cada nodo es una hoja o tiene un grado exactamente 2. No hay nodos de grado 1. En consecuencia, el orden de los hijos de un nodo conserva la información de posición.

Podemos extender la información de posicionamiento que distingue árboles binarios de árboles ordenados a árboles con más de 2 hijos por nodo. En un árbol posicional, el

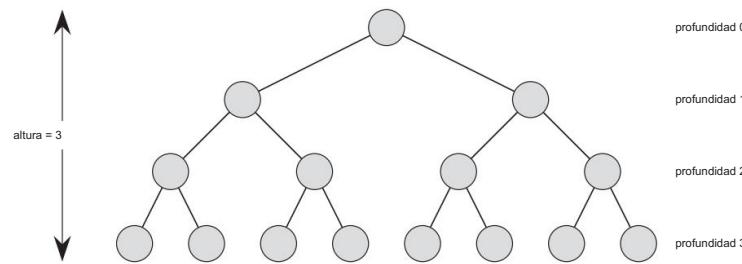


Figura B.8 Un árbol binario completo de altura 3 con 8 hojas y 7 nodos internos.

los hijos de un nodo están etiquetados con enteros positivos distintos. El hijo  $i$ -ésimo de un nodo está ausente si ningún hijo está etiquetado con el entero  $i$ . Un árbol  $k$ -ario es un árbol posicional en el que, para cada nodo, faltan todos los elementos secundarios con etiquetas mayores que  $k$ . Por tanto, un árbol binario es un árbol  $k$ -ario con  $k = 2$ .

Un árbol k-ario completo es un árbol k-ario en el que todas las hojas tienen la misma profundidad y todos los nodos internos tienen grado k. La figura B.8 muestra un árbol binario completo de altura 3. ¿Cuántas hojas tiene un árbol k-ario completo de altura h? La raíz tiene k hijos en la profundidad 1, cada uno de los cuales tiene k hijos en la profundidad 2, etc. Por lo tanto, el número de hojas en la profundidad h es  $kh$ . En consecuencia, la altura de un árbol k-ario completo con n hojas es  $\log_k n$ . El número de nodos internos de un árbol k-ario completo de altura h es

por la ecuación (A.5). Así, un árbol binario completo tiene  $2^h - 1$  nodos internos.

Fiercicios

B 5-1

Dibuja todos los árboles libres compuestos por los tres vértices  $x$ ,  $y$  y  $y'$ . Dibuja todos los árboles enraizados con nodos  $x$ ,  $y$  y  $y'$  con  $x$  como raíz. Dibuja todos los árboles ordenados con nodos  $x$ ,  $y$  y  $y'$  con  $x$  como raíz. Dibuja todos los árboles binarios con nodos  $x$ ,  $y$  y  $y'$  con  $x$  como raíz.

**B.5-2**

Sea  $G = \langle V, E \rangle$ . Sea un grafo acíclico dirigido en el que hay un vértice  $0 \in V$  tal que existe un único camino desde a cada vértice  $v \in V$ . Demuestre que la versión no dirigida de  $G$  forma un árbol.

**B.5-3**

Muestre por inducción que el número de nodos de grado 2 en cualquier árbol binario no vacío es 1 menos que el número de hojas. Concluya que el número de nodos internos en un árbol binario completo es 1 menos que el número de hojas.

**B.5-4**

Utilice la inducción para demostrar que un árbol binario no vacío con  $n$  nodos tiene una altura de al menos  $\lg n$ .

**B.5-5 ?**

La longitud de la ruta interna de un árbol binario completo es la suma, tomada de todos los nodos internos del árbol, de la profundidad de cada nodo. Asimismo, la longitud del camino externo es la suma, sobre todas las hojas del árbol, de la profundidad de cada hoja. Considere un árbol binario completo con  $n$  nodos internos, longitud de ruta interna  $i$  y longitud de ruta externa  $e$ .

Demuestre que  $e \leq i + Cn$ .

**B.5-6 ?**

Asociemos un "peso"  $w_x/D$  con cada hoja  $x$  de profundidad  $d$  en un árbol binario  $T$ .  $w_x/D$  se conoce como la profundidad de hoja de  $T$ . Demuestre que  $\sum_{x \in T} w_x \geq \frac{1}{D}$ .

**B.5-7 ?**

Muestre que si  $L \geq 2$ , entonces todo árbol binario con  $L$  hojas contiene un subárbol que tiene entre  $L=3$  y  $2L=3$  hojas, inclusive.

**Problemas****B-1 Coloración de grafos**

Dado un grafo no dirigido  $G = \langle V, E \rangle$ , una  $k$ -coloración de  $G$  es una función  $c: V \rightarrow \{0, 1, \dots, k-1\}$  tal que  $c(u) \neq c(v)$  para toda arista  $uv \in E$ . En otras palabras, los números  $0, 1, \dots, k-1$  representan los  $k$  colores, y los vértices adyacentes deben tener colores diferentes.

a. Demuestre que cualquier árbol es de 2 colores.

b. Demuestre que los siguientes son equivalentes:

1. G es bipartito.
2. G es de 2 colores.
3. G no tiene ciclos de duración impar.

C. Sea  $d$  el grado máximo de cualquier vértice en un grafo  $G$ . Demostrar que podemos color  $G$  con  $\lceil \frac{d}{2} \rceil + 1$  colores.

d. Demuestre que si  $G$  tiene aristas  $O(|V|)$ , entonces podemos colorear  $G$  con colores  $O(\sqrt{|V|})$ .

### B-2 Gráficas amigables

Reformule cada uno de los siguientes enunciados como un teorema sobre gráficas no dirigidas y luego demuéstrelo. Suponga que la amistad es simétrica pero no reflexiva.

- a. Cualquier grupo de al menos dos personas contiene al menos dos personas con el mismo número de amigos en el grupo.
- b. Cada grupo de seis personas contiene al menos tres amigos en común o al menos tres extraños en común.
- c. Cualquier grupo de personas se puede dividir en dos subgrupos de modo que al menos la mitad de los amigos de cada persona pertenezcan al subgrupo del que esa persona no es miembro.
- d. Si todos en un grupo son amigos de al menos la mitad de las personas del grupo, entonces el grupo puede sentarse alrededor de una mesa de tal manera que todos estén sentados entre dos amigos.

### B-3 Bisección de árboles

Muchos algoritmos de divide y vencerás que operan en gráficos requieren que el gráfico se divida en dos subgráficos de tamaño casi igual, que son inducidos por una partición de los vértices. Este problema investiga las bisecciones de árboles formadas al eliminar una pequeña cantidad de bordes. Requerimos que cada vez que dos vértices terminen en el mismo subárbol después de eliminar los bordes, deben estar en la misma partición.

- a. Muestre que podemos dividir los vértices de cualquier árbol binario de  $n$ -vértices en dos  $3n=4$ , conjuntos  $A$  y  $B$ , tales que  $|A|=3n=4$  y  $|B|=n$  arista. eliminando un solo
- b. Muestre que la constante  $3=4$  en la parte (a) es óptima en el peor de los casos dando un ejemplo de un árbol binario simple cuya partición balanceada más uniformemente al eliminar un solo borde tiene  $|A|=3n=4$ .

C. Demuestre que eliminando como máximo  $O.lg n$  aristas, podemos dividir los vértices de cualquier árbol binario de  $n$ -vértices en dos conjuntos A y B tales que  $|A| D bn=2c$  y  $|B| D dn=2e$ .

---

#### Notas del apéndice

G. Boole fue pionero en el desarrollo de la lógica simbólica e introdujo muchas de las notaciones de conjuntos básicas en un libro publicado en 1854. G. Cantor creó la teoría de conjuntos moderna durante el período 1874-1895. Cantor se centró principalmente en conjuntos de cardinalidad infinita. El término “función” se atribuye a GW Leibniz, quien lo utilizó para referirse a varios tipos de fórmulas matemáticas. Su definición limitada se ha generalizado muchas veces. La teoría de grafos se originó en 1736, cuando L. Euler demostró que era imposible cruzar cada uno de los siete puentes de la ciudad de Königsberg exactamente una vez y volver al punto de partida.

El libro de Harary [160] proporciona un compendio útil de muchas definiciones y resultados de la teoría de grafos.

---

## C Conteо y probabilidad

Este apéndice repasa la combinatoria elemental y la teoría de la probabilidad. Si tiene una buena experiencia en estas áreas, puede hojear ligeramente el principio de este apéndice y concentrarse en las secciones posteriores. La mayoría de los capítulos de este libro no requieren probabilidad, pero para algunos capítulos es esencial.

La sección C.1 revisa los resultados elementales de la teoría del conteo, incluidas las fórmulas estándar para contar permutaciones y combinaciones. Los axiomas de probabilidad y los hechos básicos relacionados con las distribuciones de probabilidad forman la Sección C.2. Las variables aleatorias se introducen en la Sección C.3, junto con las propiedades de la expectativa y la varianza. La sección C.4 investiga las distribuciones geométricas y binomiales que surgen del estudio de los ensayos de Bernoulli. El estudio de la distribución binomial continua en la Sección C.5, una discusión avanzada de las “colas” de la distribución.

---

### C.1 Conteо

La teoría del conteo intenta responder a la pregunta “¿Cuántos?” sin enumerar realmente todas las opciones. Por ejemplo, podríamos preguntar: “¿Cuántos números de  $n$  bits diferentes hay?” o “¿Cuántos ordenamientos de  $n$  elementos distintos hay?” En esta sección, revisamos los elementos de la teoría del conteo. Dado que parte del material supone una comprensión básica de los conjuntos, es posible que desee comenzar revisando el material de la Sección B.1.

Reglas de suma y producto.

A veces podemos expresar un conjunto de elementos que deseamos contar como una unión de conjuntos disjuntos o como un producto cartesiano de conjuntos.

La regla de la suma dice que el número de formas de elegir un elemento de uno de dos conjuntos disjuntos es la suma de las cardinalidades de los conjuntos. Es decir, si  $A$  y  $B$  son dos conjuntos finitos sin miembros en común, entonces  $|A \cup B| = |A| + |B|$ , que

se sigue de la ecuación (B.3). Por ejemplo, cada posición en la matrícula de un automóvil es una letra o un dígito. Por lo tanto, el número de posibilidades para cada posición es 26 C 10 D 36, ya que hay 26 opciones si es una letra y 10 opciones si es un dígito.

La regla del producto dice que el número de formas de elegir un par ordenado es el número de formas de elegir el primer elemento multiplicado por el número de formas de elegir el segundo elemento. Es decir, si A y B son dos conjuntos finitos, entonces  $jA \cdot jB = j(A \times B)$ , que es simplemente la ecuación (B.4). Por ejemplo, si una heladería ofrece 28 sabores de helado y 4 aderezos, el número de sundaes posibles con una bola de helado y un aderezo es 28 C 4 D 112.

#### Instrumentos de cuerda

Una cadena sobre un conjunto finito S es una secuencia de elementos de S. Por ejemplo, hay 8 cadenas binarias de longitud 3:

000; 001; 010; 011; 100; 101; 110; 111

A veces llamamos a una cadena de longitud k una k-cadena. Una subcadena  $s_0$  de una cadena s es una secuencia ordenada de elementos consecutivos de s. Una k-subcadena de una cadena es una subcadena de longitud k. Por ejemplo, 010 es una subcadena de longitud 3 de 01101001 (la subcadena de longitud 3 que comienza en la posición 4), pero 111 no es una subcadena de 01101001.

Podemos ver una k-cadena sobre un conjunto S como un elemento del producto cartesiano  $S^k$  de k-tuplas; por tanto, hay  $jS^k$  cadenas de longitud k. Por ejemplo, el número de k-cadenas binarias es  $2^k$ . Intuitivamente, para construir una k-cadena sobre un n-conjunto, tenemos n formas de elegir el primer elemento; para cada una de estas opciones, tenemos n formas de elegir el segundo elemento; y así sucesivamente k veces. Esta construcción conduce al producto de k veces nn...n D  $n^k$  como el número de k-cadenas.

#### permutaciones

Una permutación de un conjunto finito S es una secuencia ordenada de todos los elementos de S, donde cada elemento aparece exactamente una vez. Por ejemplo, si  $S = \{a, b, c\}$ , entonces S tiene 6 permutaciones:

a B C; acb; bac; bca; taxi; cba

Hay  $n!$  permutaciones de un conjunto de n elementos, ya que podemos elegir el primer elemento de la sucesión de n formas, el segundo de  $n - 1$  formas, el tercero de  $n - 2$  formas, y así sucesivamente.

Una k-permutación de S es una secuencia ordenada de k elementos de S, sin que ningún elemento aparezca más de una vez en la secuencia. (Así, una permutación ordinaria es una n-permutación de un n-conjunto.) Las doce 2-permutaciones del conjunto  $\{a, b, c, d\}$  son

ab; C.A; anuncio; licenciado en Letras; antes de Cristo; bd; California; cb; cd; da; base de datos; corriente continua

El número de k-permutaciones de un n-conjunto es

$$\frac{n!}{n-k} = \frac{n(n-1)(n-2)\dots(n-k+1)}{k!} \quad (C.1)$$

ya que tenemos n formas de elegir el primer elemento, n-1 formas de elegir el segundo elemento, y así sucesivamente, hasta que hayamos seleccionado k elementos, siendo el último una selección de los nk C 1 elementos restantes.

combinaciones

Una k-combinación de un n-conjunto S es simplemente un k-subconjunto de S. Por ejemplo, el 4-conjunto {a; b; C; dg} tiene seis combinaciones de 2:

ab; C.A; anuncio; antes de Cristo; bd; cd

(Aquí usamos la forma abreviada de denotar el 2-subconjunto {a; b} por ab, y así sucesivamente).

Podemos construir una combinación k de un conjunto n eligiendo k elementos distintos (diferentes) del conjunto n. No importa el orden en que seleccionemos los elementos.

Podemos expresar el número de k-combinaciones de un n-conjunto en términos del número de k-permutaciones de un n-conjunto. Cada k-combinación tiene exactamente k! permutaciones de sus elementos, cada uno de los cuales es una k-permutación distinta del n-conjunto. Así, el número de k-combinaciones de un n-conjunto es el número de k-permutaciones dividido por k!, de la ecuación (C.1), esta cantidad es

$$\frac{n!}{k!(n-k)!} \quad (C.2)$$

Para k=0, esta fórmula nos dice que el número de formas de elegir 0 elementos de un conjunto n es 1 (no 0), ya que  $0! = 1$ .

Coefficientes binomiales

La notación (*léase " $\binom{n}{k}$  elige k"*) denota el número de k-combinaciones de un n-conjunto. De la ecuación (C.2), tenemos

$$\frac{n!}{k!(n-k)!}$$

Esta fórmula es simétrica en k y nk:

$$\frac{n!}{k!(n-k)!} = \frac{(nk)!}{(nk-k)!k!} \quad (C.3)$$

$k! \neq nk!$

Estos números también se conocen como coeficientes binomiales, debido a su aparición en la expansión binomial:

$$\frac{x^n}{k!} \binom{n}{k} = \frac{x^k}{k!} \cdot \frac{(n-k)!}{(n-k)!} \quad (C.4)$$

Un caso especial de la expansión binomial ocurre cuando  $x = D$  y  $D = 1$ :

$$\frac{D^n}{k!} \binom{n}{k} = \frac{1^n}{k!} \binom{n}{k}$$

Esta fórmula corresponde a contar las  $2^n$  cadenas de  $n$  binarias por el número de 1 que contienen: las  $\binom{n}{k}$  cadenas de  $n$  binarias contienen exactamente  $k$  1, ya que tenemos  $\binom{n}{k}$  formas de elegir  $k$  de las  $n$  posiciones en las que colocar los 1.

Muchas identidades involucran coeficientes binomiales. Los ejercicios al final de esta sección le dan la oportunidad de probar algunos.

### Límites binomiales

A veces necesitamos acotar el tamaño de un coeficiente binomial. Para  $1 \leq k \leq n$ , tenemos el límite inferior

$$\frac{\frac{n(n-1)\dots(n-k+1)}{k!}}{\frac{1}{k!}} = \frac{n}{k} \cdot \frac{n-1}{k-1} \cdots \frac{n-k+1}{1}$$

Stirling Aprovechando la desigualdad  $\sqrt{2\pi n} \leq \Gamma(n+1) \leq \sqrt{4\pi n} e^{-(n+1/2)}$  derivada del aproximado de Stirling (3.18), obtenemos las cotas superiores

$$\frac{\frac{n(n-1)\dots(n-k+1)}{k!}}{\frac{1}{k!}} < \frac{\sqrt{2\pi n} \cdot \sqrt{2\pi(n-1)} \cdots \sqrt{2\pi(n-k+1)}}{\frac{1}{k!}} \cdot \frac{e^{-n} \cdot e^{-(n-1)} \cdots e^{-(n-k+1)}}{\frac{1}{k!}} \quad (C.5)$$

Para todos los enteros  $k$  tales que  $0 \leq k \leq n$ , podemos usar la inducción (vea el Ejercicio C.1-12) para probar el límite

$$\frac{n!}{k!} \frac{n^n}{k^n k^{n-k}} \quad : \quad (C.6)$$

donde por conveniencia asumimos que  $0^0 = 1$ . Para  $k \leq n$ , donde 0 puede reescribir este límite como

$$\frac{n^n \cdot n/n \cdot 1/n \cdot 1/n}{n!} \\ = \frac{1}{1 - \frac{1}{1}} = \frac{1}{1 - e^{-H_0}} \quad ;$$

$H_0 = \sum_{i=1}^n p_i \lg p_i$

1, nosotros

dónde

$$H_0 = \sum_{i=1}^n p_i \lg p_i \quad ; \quad (C.7)$$

es la función de entropía (binaria) y donde, por conveniencia, asumimos que  $0 \lg 0 = 0$ , por lo que  $H_0 = \sum_{i=1}^n p_i \lg p_i$ .

### Ejercicios

#### C.1-1

¿Cuántas  $k$ -subcadenas tiene una  $n$ -cadena? (Considere que las  $k$ -subcadenas idénticas en diferentes posiciones son diferentes). ¿Cuántas subcadenas tiene una  $n$ -cadena en total?

#### C.1-2

Una función booleana de  $n$  entradas y  $m$  salidas es una función de fTRUE; FALSO y VERDADERO; FALSO ej. m. ¿Cuántas funciones booleanas de  $n$  entrada y 1 salida hay?

¿Cuántas funciones booleanas de entrada  $n$  y salida  $m$  hay?

#### C.1-3

¿De cuántas maneras pueden sentarse  $n$  profesores alrededor de una mesa de conferencias circular? Considere que dos asientos son iguales si uno se puede girar para formar el otro.

#### C.1-4

¿ De cuántas maneras podemos elegir tres números distintos del conjunto  $\{1; 2; \dots; 99\}$  para que su suma sea par?

C.1-5

Acreditar la identidad

$$\frac{n!}{k!} D^{\overline{k}} \stackrel{\text{norte}}{=} \frac{n-1}{k-1}!$$
(C.8)

para  $0 < k < n$ .

C.1-6

Acreditar la identidad

$$\frac{n!}{k!} D^{\overline{nk}} \stackrel{\text{norte}}{=} \frac{n-1}{k}$$

para  $0 < k < n$ .

C.1-7

Para elegir  $k$  objetos de  $n$ , puede distinguir uno de los objetos y considerar si se elige el objeto distinguido. Utilice este enfoque para demostrar que

$$\frac{n!}{k!} D^{\overline{k}} \stackrel{\text{norte}}{=} \frac{n-1}{k-1} ! :$$

C.1-8

el resultado del ejercicio C.1-7, haga una tabla para  $n! D^{\overline{0}}, 1, \dots, 6$  y  $0 \leq k \leq n$ . Usando coeficientes binomiales con en la parte superior, en la línea siguiente, y así sucesivamente. Tal tabla de coeficientes binomiales se llama triángulo de Pascal.

C.1-9

Demostrar que

$$\frac{n!}{k!} D^{\overline{k}} \stackrel{\text{idem}}{=} \frac{n-1}{k-1} ! :$$

C.1-10

Muestre que para cualquier número entero  $n \geq 0$  y  $0 \leq k \leq n$ , la expresión  $\frac{n!}{k!}$  logra su valor máximo cuando  $k = bn=2c$  o  $k = dn=2e$ .

C.1-11 ?

Argumente que para cualquier número entero  $n \geq 0$ ,  $j \geq 0$ ,  $k \geq 0$  y  $j \leq k \leq n$ ,

$$\frac{n!}{j! k!} \stackrel{\text{norte}}{=} \frac{n-j}{n-k} ! : \quad (C.9)$$

Proporcione tanto una prueba algebraica como un argumento basado en un método para elegir  $j$  elementos de  $n$ . Da un ejemplo en el que no se cumpla la igualdad.

C.1-12 ?

Usa la inducción en todos los enteros  $k$  tales que  $0 \leq k \leq n$  para probar la desigualdad (C.6), y usa la ecuación (C.3) para extenderla a todos los enteros  $k$  tales que  $0 \leq k < n$ .

C.1-13 ?

Use la aproximación de Stirling para demostrar que

$$\frac{2^n}{n!} \geq \frac{2^{2n}}{p^n} \cdot 1 > 0 \quad \text{C.1-13 :} \quad (\text{C.10})$$

C.1-14 ?

Derivando la función de entropía  $H$ , demuestre que alcanza su valor máximo en  $D = 1/2$ . ¿Qué es  $H(1/2)$ ?

C.1-15 ?

Demuestre que para cualquier entero  $n \geq 0$ ,

$$\sum_{k=0}^n \frac{x_n}{k!} \leq e^{x_n} \quad (\text{C.11})$$

## C.2 Probabilidad

La probabilidad es una herramienta esencial para el diseño y análisis de algoritmos probabilísticos y aleatorios. Esta sección repasa la teoría básica de la probabilidad.

Definimos la probabilidad en términos de un espacio muestral  $S$ , que es un conjunto cuyos elementos se denominan eventos elementales. Podemos pensar en cada evento elemental como un posible resultado de un experimento. Para el experimento de lanzar dos monedas distinguibles, con cada lanzamiento individual dando como resultado una cara (H) o una cruz (T), podemos ver el espacio muestral como el conjunto de todas las 2 cuerdas posibles sobre fH; Tg:

SD fHH; HT; TH; TTg :

Un evento es un subconjunto<sup>1</sup> del espacio muestral S. Por ejemplo, en el experimento de lanzar dos monedas, el evento de obtener una cara y una cruz es fHT; THg. El evento S se llama evento cierto, y el evento ; se llama el evento nulo. Decimos que dos eventos A y B son mutuamente excluyentes si A\BD;. A veces tratamos un evento elemental s 2 S como el evento fsg. Por definición, todos los eventos elementales son mutuamente excluyentes.

### Axiomas de probabilidad

Una distribución de probabilidad Pr fg en un espacio muestral S es un mapeo de eventos de S a números reales que satisfacen los siguientes axiomas de probabilidad:

1. Pr fAg 0 para cualquier evento A.
2. Pr fSg D 1.
3. Pr fA [ Bg D Pr fAg C Pr fBg para dos eventos A y B mutuamente excluyentes cualesquiera. Más generalmente, para cualquier secuencia (finita o numerable infinita) de eventos A1; A2;::: que son mutuamente excluyentes por pares,

$$\Pr \text{ fai } g : \\ \Pr ( [ i \in A_i ) \rightarrow \sum_i$$

Llamamos Pr fAg a la probabilidad del evento A. Notamos aquí que el axioma 2 es un requisito de normalización: en realidad no hay nada fundamental en elegir 1 como la probabilidad de cierto evento, excepto que es natural y conveniente.

Varios resultados se derivan inmediatamente de estos axiomas y la teoría básica de conjuntos (ver Sección B.1). El evento nulo; tiene probabilidad Pr f;g D 0. Si AB, entonces Pr fAg Pr fBg. Usando A para denotar el evento SA (el complemento de A), tenemos Pr fAD 1 Pr fAg. Para cualesquiera dos eventos A y B,

$$\Pr fA [ Bg D \Pr fAg C \Pr fBg \Pr fA \setminus Bg \quad (C.12)$$

$$\Pr fAg C \Pr fBg : \quad (C.13)$$

<sup>1</sup>Para una distribución de probabilidad general, puede haber algunos subconjuntos del espacio muestral S que no se consideran eventos. Esta situación generalmente surge cuando el espacio muestral es incontablemente infinito. El principal requisito para que los subconjuntos sean eventos es que el conjunto de eventos de un espacio muestral sea cerrado bajo las operaciones de tomar el complemento de un evento, formar la unión de un número finito o contable de eventos y tomar la intersección de un finito o número contable de eventos. La mayoría de las distribuciones de probabilidad que veremos son sobre espacios muestrales finitos o contables y, en general, consideraremos que todos los subconjuntos de un espacio muestral son eventos. Una excepción notable es la distribución de probabilidad uniforme continua, que veremos en breve.

En nuestro ejemplo de lanzar una moneda, suponga que cada uno de los cuatro eventos elementales tiene probabilidad 1=4. Entonces la probabilidad de obtener al menos una cara es

$$\Pr\{fHH\} ; \Pr\{HT\} ; \Pr\{TH\} ; \Pr\{TT\}$$

re 3=4 :

Alternativamente, dado que la probabilidad de obtener estrictamente menos de una cara es  $\Pr\{TT\}$  D 1=4, la probabilidad de obtener al menos una cara es 1 1=4 D 3=4.

#### Distribuciones de probabilidad discretas

Una distribución de probabilidad es discreta si se define sobre un espacio muestral finito o contablemente infinito. Sea  $S$  el espacio muestral. Entonces para cualquier evento  $A$ ,

$$\Pr\{A\} = \sum_{s \in A} p_s$$

ya que los eventos elementales, específicamente los de  $A$ , son mutuamente excluyentes. Si  $S$  es finito y todo evento elemental  $s \in S$  tiene probabilidad

$$\Pr\{s\} = p_s$$

entonces tenemos la distribución de probabilidad uniforme en  $S$ . En tal caso, el experimento a menudo se describe como "elegir un elemento de  $S$  al azar".

Como ejemplo, considere el proceso de lanzar una moneda justa, una para la cual la probabilidad de obtener cara es la misma que la probabilidad de obtener cruz, es decir, 1=2. Si lanzamos la moneda  $n$  veces, tenemos la distribución de probabilidad uniforme definido en el espacio muestral  $S = \{H, T\}^n$ . Podemos representar cada evento elemental en  $S$  como una cadena de longitud  $n$  sobre  $\{H, T\}$ , cada cadena ocurre con probabilidad 1=2 $n$ . El evento

$A$  exactamente  $k$  caras y exactamente  $n-k$  cruces ocurren

es un subconjunto de  $S$  de tamaño  $\binom{n}{k}$  cadenas de longitud  $n$  sobre  $\{H, T\}$ .  $A$  contiene exactamente  $k$  H. La probabilidad del evento  $A$  es entonces  $\Pr\{A\} = \frac{\binom{n}{k}}{2^n}$ .

$n$

#### Distribución de probabilidad uniforme continua

La distribución de probabilidad uniforme continua es un ejemplo de una distribución de probabilidad en la que no todos los subconjuntos del espacio muestral se consideran eventos. La distribución de probabilidad uniforme continua se define sobre un intervalo cerrado  $[a, b]$  de los reales, donde  $a < b$ . Nuestra intuición es que cada punto en el intervalo  $[a, b]$  debe ser "igualmente probable". Hay un número incontable de puntos, sin embargo, si damos a todos los puntos la misma probabilidad positiva finita, no podemos satisfacer simultáneamente los axiomas 2 y 3. Por esta razón, nos gustaría asociar un

probabilidad sólo con algunos de los subconjuntos de  $S$ , de tal forma que se satisfacen los axiomas para estos eventos.

Para cualquier intervalo cerrado  $C_E; d$ , donde  $a \leq b$ , el uniforme continuo la distribución de probabilidad define la probabilidad del evento  $C_E; d$  ser

$$\Pr f_{C_E; d} = \frac{\text{contiene continua}}{b-a}$$

Tenga en cuenta que para cualquier punto  $x \in C_E; x$ , la probabilidad de  $x$  es 0. Si quitamos los extremos de un intervalo  $C_E; d$ , obtenemos el intervalo abierto  $(c; d)$ . Desde  $C_E; d \subset C_E; c \cup (c; d) \cup C_E; d$ , el axioma 3 nos da  $\Pr f_{C_E; d} = \Pr f_{C_E; c} + \Pr f_{(c; d)} + \Pr f_{C_E; d}$ . Generalmente, el conjunto de eventos para la distribución de probabilidad uniforme continua contiene cualquier subconjunto del espacio muestral  $C_E$ ;  $b$  que se puede obtener mediante una unión finita o numerable de intervalos abiertos y cerrados, así como ciertos conjuntos más complicados.

### Probabilidad condicional e independencia

A veces tenemos algún conocimiento parcial previo sobre el resultado de un experimento. Por ejemplo, suponga que un amigo ha lanzado al aire dos monedas justas y le ha dicho que al menos una de las monedas mostró cara. ¿Cuál es la probabilidad de que ambas monedas sean cara? La información dada elimina la posibilidad de dos cruces. Los tres eventos elementales restantes son igualmente probables, por lo que inferimos que cada uno ocurre con una probabilidad de  $1/3$ . Dado que solo uno de estos eventos elementales muestra dos caras, la respuesta a nuestra pregunta es  $1/3$ .

La probabilidad condicional formaliza la noción de tener un conocimiento parcial previo del resultado de un experimento. La probabilidad condicional de un evento  $A$  dado que ocurre otro evento  $B$  se define como

$$\Pr f_{A|B} = \frac{\Pr f_{A \setminus B}}{\Pr f_B} \quad (C.14)$$

siempre que  $\Pr f_B \neq 0$ . (Leemos “ $\Pr f_{A|B}$ ” como “la probabilidad de  $A$  dado  $B$ ”).

Intuitivamente, dado que sabemos que ocurre el evento  $B$ , el evento de que  $A$  también ocurra es  $A \setminus B$ . Es decir,  $A \setminus B$  es el conjunto de resultados en los que ocurren tanto  $A$  como  $B$ .

Como el resultado es uno de los eventos elementales en  $B$ , normalizamos las probabilidades de todos los eventos elementales en  $B$  dividiéndolos por  $\Pr f_B$ , de modo que sumen 1. La probabilidad condicional de  $A$  dado  $B$  es, por lo tanto, la relación entre la probabilidad del evento  $A \setminus B$  y la probabilidad del evento  $B$ . En el ejemplo anterior,  $A$  es el evento de que ambas monedas sean cara y  $B$  es el evento de que al menos una moneda sea cara. Así,  $\Pr f_{A|B} = 1/3 = 1/2$ .

Dos eventos son independientes si

$$\Pr f_{A \setminus B} = \Pr f_A \quad (C.15)$$

que es equivalente, si  $\Pr f_B \neq 0$ , a la condición

$\Pr fA \cap Bg \leq D \Pr fAg :$

Por ejemplo, suponga que lanzamos dos monedas al aire y que los resultados son independientes. Entonces la probabilidad de dos caras es  $.1=2/.1=2/ D 1=4$ . Ahora suponga que un evento es que la primera moneda sale cara y el otro evento es que las monedas salen de manera diferente. Cada uno de estos eventos ocurre con probabilidad  $1=2$ , y la probabilidad de que ambos eventos ocurran es  $1=4$ ; por lo tanto, de acuerdo con la definición de independencia, los eventos son independientes, aunque se podría pensar que ambos eventos dependen de la primera moneda. Finalmente, suponga que las monedas están soldadas entre sí de modo que ambas caigan cara o ambas caigan cruz y que las dos posibilidades son igualmente probables. Entonces la probabilidad de que cada moneda salga cara es  $1=2$ , pero la probabilidad de que ambas salgan cara es  $1=2 \neq .1=2/.1=2/$ . En consecuencia, el evento de que uno salga cara y el evento de que el otro salga cara no son independientes.

Una colección  $A_1; A_2; \dots$ ; Se dice que un evento es independiente por pares si

$\Pr fai \setminus Aj g \leq D \Pr fai g \Pr faj g$

para todo  $1 < j n$ . Decimos que los eventos de la colección son (mutuamente) independientes si todo  $k$ -subconjunto  $A_{i1} ; A_{i2} ; \dots ; A_{ik}$  de la colección, donde  $2 kn$  y  $1 i_1 < i_2 < \dots < i_k n$ , satisface

$\Pr fai_1 \setminus Ai_2 \setminus \dots \setminus A_{ik} g \leq D \Pr fAi_1 g \Pr fai_2 g \Pr faik g :$

Por ejemplo, supongamos que lanzamos dos monedas justas. Sea  $A_1$  el evento de que la primera moneda salga cara, sea  $A_2$  el evento de que la segunda moneda salga cara y sea  $A_3$  el evento de que las dos monedas sean diferentes. Tenemos

$$\Pr fA_1 g D 1=2;$$

$$\Pr fA_2 g D 1=2;$$

$$\Pr fA_3 g D 1=2;$$

$$\Pr fA_1 \setminus A_2 g D 1 = 4 ;$$

$$\Pr fA_1 \setminus A_3 g D 1 = 4;$$

$$\Pr fA_2 \setminus A_3 g D 1 = 4;$$

$$\Pr fA_1 \setminus A_2 \setminus A_3 g D 0$$

Como para  $1 < j 3$ , tenemos  $\Pr fAi \setminus Aj g \leq D \Pr fAi g \Pr fAj g D 1=4$ , los eventos  $A_1, A_2$  y  $A_3$  son independientes por pares. Sin embargo, los eventos no son mutuamente independientes porque  $\Pr fA_1 \setminus A_2 \setminus A_3 g D 0 \neq \Pr fA_1 g \Pr fA_2 g \Pr fA_3 g D 1=8 \neq 0$ .

teorema de bayes

De la definición de probabilidad condicional (C.14) y la ley conmutativa  $A \setminus BDB \setminus A$ , se deduce que para dos eventos  $A$  y  $B$ , cada uno con probabilidad distinta de cero,  $\Pr fA \setminus Bg D \Pr fBg Pr fA j Bg D \Pr fAg Pr$

$fB j Ag$  : Resolviendo para  $\Pr fA j Bg$ , (C.16)

obtenemos

$$\frac{\Pr fAg \Pr fB j Ag Pr}{\Pr fA j Bg D \Pr fBg} ; \quad (C.17)$$

que se conoce

como el teorema de Bayes. El denominador  $\Pr fBg$  es una constante de normalización, que podemos reformular de la siguiente manera. Como  $BD .B \setminus A / [ .B \setminus A]$ , y como  $B \setminus A$  y  $B \setminus A$  son eventos mutuamente excluyentes,  $\Pr fBg D \Pr fB \setminus Ag C \Pr ^\circ B \setminus AD Pr$

$fAg Pr fB j Ag C \Pr ^\circ A Pr ^\circ B j A$  Sustituyendo

en la ecuación (C.17), obtenemos una forma equivalente de la teoría de Bayes

$$\frac{\Pr fAg \Pr fB j Ag Pr}{\Pr fA j Bg D \Pr fAg} = \frac{\Pr fB j Ag C \Pr ^\circ A Pr ^\circ B j A}{\Pr fB j Ag C \Pr ^\circ A Pr ^\circ B j A} \quad (C.18)$$

El teorema de

Bayes puede simplificar el cálculo de probabilidades condicionales. Por ejemplo, supongamos que tenemos una moneda justa y una moneda sesgada que siempre sale cara. Realizamos un experimento que consta de tres eventos independientes: elegimos una de las dos monedas al azar, lanzamos esa moneda una vez y luego la lanzamos de nuevo.

Supongamos que la moneda que hemos elegido sale cara las dos veces. ¿Cuál es la probabilidad de que esté sesgado?

Resolvemos este problema usando el teorema de Bayes. Sea  $A$  el evento de que elegimos la moneda sesgada, y sea  $B$  el evento de que la moneda elegida salga cara en ambas ocasiones. Deseamos determinar  $\Pr fA j Bg$ . Tenemos  $\Pr fAg D 1=2$ ,  $\Pr fB j Ag D 1$ ,  $\Pr ^\circ AD 1=2$  y  $\Pr ^\circ B j AD 1=4$ ; por lo tanto,  $.1=2/ 1 .1=2/ 1 C .1=2/ .1=4/ D 4=5$  :

$$\Pr fA j Bg D$$

## Ejercicios

### C.2-1

El profesor Rosencrantz lanza una moneda al aire una vez. El profesor Guildenstern lanza una moneda justa dos veces. ¿Cuál es la probabilidad de que el profesor Rosencrantz obtenga más cabezas que el profesor Guildenstern?

C.2-2

Demostrar la desigualdad de Boole: Para cualquier secuencia finita o numerable infinita de eventos  $A_1; A_2; \dots;$ ,

$$\Pr f{A_1} \wedge A_2 \mid g \Pr f{A_1} \wedge C \Pr f{A_2} \wedge C : \quad (C.19)$$

C.2-3

Supongamos que barajamos una baraja de 10 cartas, cada una con un número distinto del 1 al 10, para mezclar bien las cartas. Luego retiramos tres cartas, una a la vez, de la baraja.

¿Cuál es la probabilidad de que seleccionemos las tres cartas en orden ordenado (creciente)?

C.2-4

Demostrar que

$$\Pr f{A_j \wedge B_g} \leq \Pr f{\overline{A_j} \wedge \overline{B_D}} \quad 1$$

C.2-5

Demostrar que para cualquier colección de eventos  $A_1; A_2; \dots; A_n,$

$$\Pr f{A_1 \wedge A_2 \wedge \dots \wedge A_n} \leq \Pr f{A_1} \Pr f{A_2} \dots \Pr f{A_n}$$

$$\Pr f{\text{fan } j \text{ } A_1 \wedge A_2 \wedge \dots \wedge A_n} :$$

C.2-6 ?

Describa un procedimiento que tome como entrada dos números enteros  $a$  y  $b$  tales que  $0 < a < b$  y, al lanzar una moneda al aire, produzca como salida cara con probabilidad  $a=b$  y cruz con probabilidad  $.ba/b = b$ . Proporcione un límite para el número esperado de lanzamientos de moneda, que debe ser  $0.1/$ . (Pista: Representa  $a=b$  en binario).

C.2-7 ?

Muestre cómo construir un conjunto de  $n$  eventos que son independientes por pares pero tales que ningún subconjunto de  $k > 2$  de ellos es mutuamente independiente.

C.2-8 ?

Dos eventos  $A$  y  $B$  son condicionalmente independientes, dado  $C$ , si

$$\Pr f{A \wedge B \mid C} = \Pr f{A \mid C} \Pr f{B \mid C}$$

Dé un ejemplo simple pero no trivial de dos eventos que no son independientes pero que son condicionalmente independientes dado un tercer evento.

C.2-9 ?

Eres un concursante en un programa de juegos en el que se esconde un premio detrás de una de las tres cortinas. Ganarás el premio si seleccionas la cortina correcta. Después de usted

ha elegido un telón, pero antes de que se levante el telón, el maestro de ceremonias levanta uno de los otros telones, sabiendo que revelará un escenario vacío, y le pregunta si desea cambiar de su selección actual al telón restante. ¿Cómo cambiarían tus posibilidades si cambias? (Esta pregunta es el célebre problema de Monty Hall, llamado así por un presentador de un programa de juegos que a menudo presentaba a los concursantes este mismo dilema).

#### C.2-10 ?

El director de una prisión ha elegido al azar a un preso entre tres para que quede libre. Los otros dos serán ejecutados. El guardia sabe cuál quedará libre, pero tiene prohibido dar a cualquier prisionero información sobre su estado. Llámese a los presos y a Z. El preso X le pregunta al guardia en privado cuál de Y o Z será ejecutado X, Y, argumentando como ya sabe que al menos uno de ellos debe morir, el guardia no revelará ninguno. información sobre su propio estado. El guardia le dice a X que Y debe ser ejecutado. El prisionero X se siente más feliz ahora, ya que cree que él o el prisionero Z quedarán libres, lo que significa que su probabilidad de quedar libre ahora es 1=2. ¿Tiene razón o sus posibilidades siguen siendo 1=3? Explicar.

### C.3 Variables aleatorias discretas

Una variable aleatoria (discreta) X es una función de un espacio muestral S finito o contablemente infinito a los números reales. Asocia un número real a cada posible resultado de un experimento, lo que nos permite trabajar con la distribución de probabilidad inducida sobre el conjunto de números resultante. Las variables aleatorias también se pueden definir para espacios muestrales incontablemente infinitos, pero plantean problemas técnicos que no es necesario abordar para nuestros propósitos. De ahora en adelante, supondremos que las variables aleatorias son discretas.

Para una variable aleatoria X y un número real x, definimos el evento  $X \in x$  como  $\{s \in S : X(s) = x\}$ ; de este modo,

$$\Pr(X \in x) = \Pr_{S^X}(\{s \in S : X(s) = x\})$$

La función

$$f(x) = \Pr(X \in x)$$

es la función de densidad de probabilidad de la variable aleatoria X. De los axiomas de probabilidad,  $\Pr(X \in x) \geq 0$  y  $\sum_x \Pr(X \in x) = 1$ .

Como ejemplo, considere el experimento de lanzar un par de dados ordinarios de 6 caras. Hay 36 eventos elementales posibles en el espacio muestral. Asumimos

que la distribución de probabilidad es uniforme, de modo que cada evento elemental s 2 S es igualmente probable:  $\Pr f_{\text{sg}} D = 1/36$ . Defina la variable aleatoria X como el máximo de los dos valores que se muestran en los dados. Tenemos  $\Pr f_X D = 1/36$ , ya que X asigna un valor de 3 a 5 de los 36 eventos elementales posibles, a saber,  $.1; 3/ .2; 3/ .3; 3/ .3; 2/$ , y  $.3/ .1/$ .

A menudo definimos varias variables aleatorias en el mismo espacio muestral. Si X e Y son variables aleatorias, la función

$$f(x, y) = \Pr f_X D = x \quad \Pr f_Y D = y$$

es la función de densidad de probabilidad conjunta de X e Y. Para un valor fijo y,

$$\Pr f_Y D = \sum_x \Pr f_X D = x \quad \Pr f_X D = x \quad \Pr f_Y D = y$$

y de manera similar, para un valor fijo x,

$$\Pr f_X D = x \quad \Pr f_X D = x \quad \Pr f_Y D = y$$

Usando la definición (C.14) de probabilidad condicional, tenemos

$$\Pr f_X D = x \quad \Pr f_X D = x \quad \Pr f_Y D = y$$

Definimos dos variables aleatorias X e Y como independientes si para todo x e y, los eventos  $XD = x$  e  $YD = y$  son independientes o, de manera equivalente, si para todo x e y, tenemos  $\Pr f_X D = x \quad \Pr f_Y D = y = \Pr f_X D = x \Pr f_Y D = y$ .

Dado un conjunto de variables aleatorias definidas sobre el mismo espacio muestral, podemos definir nuevas variables aleatorias como sumas, productos u otras funciones de las variables originales.

#### Valor esperado de una variable aleatoria

El resumen más simple y útil de la distribución de una variable aleatoria es el “promedio” de los valores que toma. El valor esperado (o, como sinónimo, expectativa o media) de una variable aleatoria discreta X es

$$E(X) = \sum_x x \Pr f_X D = x \quad (C.20)$$

que está bien definida si la suma es finita o converge absolutamente. A veces, la expectativa de X se denota por  $\bar{X}$  o, cuando la variable aleatoria es evidente por el contexto, simplemente por  $\bar{x}$ .

Considere un juego en el que lanza dos monedas justas. Ganas \$3 por cada cara pero pierdes \$2 por cada cruz. El valor esperado de la variable aleatoria X que representa

tus ganancias son

$$\begin{aligned} E \infty X D 6 \Pr f2 H'sg C 1 \Pr f1 H, 1 Tg 4 \Pr f2 T'sg D \\ 6.1=4/ C 1.1=2/ 4.1=4/ D 1 \end{aligned}$$

La expectativa de la suma de dos variables aleatorias es la suma de sus expectativas, es decir,

$$E \infty X CYD E \infty X CE \infty Y ; \quad (C.21)$$

siempre que se definan  $E \infty X$  y  $E \infty Y$ . Llamamos a esta propiedad linealidad de la expectativa, y se cumple incluso si  $X$  e  $Y$  no son independientes. También se extiende a sumas de expectativas finitas y absolutamente convergentes. La linealidad de la expectativa es la propiedad clave que nos permite realizar análisis probabilísticos utilizando variables aleatorias indicadoras (consulte la Sección 5.2).

Si  $X$  es cualquier variable aleatoria, cualquier función  $g_x/$  define una nueva variable aleatoria  $gX/$ . Si la expectativa de  $gX/$  está definida, entonces

$$\text{Ej. } X / DX g_x/ \Pr fX D xg :$$

Haciendo  $g_x/ D ax$ , tenemos para cualquier constante  $a$ ,

$$E \infty aX D aE \infty X : \quad (C.22)$$

En consecuencia, las expectativas son lineales: para cualquier par de variables aleatorias  $X$  e  $Y$  y cualquier constante  $a$ ,

$$E \infty aX CYD aE \infty X CE \infty Y : \quad (C.23)$$

Cuando dos variables aleatorias  $X$  e  $Y$  son independientes y cada una tiene una expectativa definida,

$$E \infty XY DX \quad \underset{x}{X} \underset{y}{xy} \Pr fX D x y YD yg$$

$$DX \quad \underset{x}{X} \underset{y}{xy} \Pr fX D xg \Pr fY D yg$$

$$\underset{x}{DX} \quad \underset{x}{x} \Pr fX D xg ! \underset{y}{X} \underset{y}{y} \Pr fY D yg !$$

$$DE \infty X E \infty Y :$$

En general, cuando  $n$  variables aleatorias  $X_1; X_2; \dots; X_n$  son mutuamente independientes,

$$E \infty X_1 X_2 \dots X_n DE \infty X_1 E \infty X_2 E \infty X_n : \quad (C.24)$$

Cuando una variable aleatoria X toma valores del conjunto de números naturales ND f0; 1; 2; : : : g, tenemos una buena fórmula para su expectativa:

$$\begin{aligned}
 E \langle EX \rangle &= \sum_{i=0}^{\infty} i \Pr fX_i \\
 X_1 &= \sum_{i=0}^{\infty} i \Pr fX_i \\
 X_1 &= \sum_{i=0}^{\infty} \Pr fX_i ; \tag{C.25}
 \end{aligned}$$

ya que cada término  $\Pr fX_i$  se suma  $i$  veces y se resta  $i - 1$  veces (excepto  $\Pr fX_0$ , que se suma 0 veces y no se resta en absoluto).

Cuando aplicamos una función convexa  $f .x/$  a una variable aleatoria X, la igualdad de Jensen nos da

$$E \langle Ef .X / f .E \langle EX \rangle \rangle ; \tag{C.26}$$

siempre que las expectativas existan y sean finitas. (Una función  $f .x/$  es convexa si para todo  $x$  e  $y$  y para todo  $0 < t < 1$ , tenemos  $f .tx + (1-t)y/ \leq tf .x/ + (1-t)f .y/$ .)

#### Varianza y desviación estándar

El valor esperado de una variable aleatoria no nos dice qué tan “dispersos” están los valores de la variable. Por ejemplo, si tenemos variables aleatorias X e Y para las cuales  $\Pr fX = 1 = 4g$  y  $\Pr fY = 0g$ , entonces tanto  $E \langle EX \rangle$  como  $E \langle EY \rangle$  son  $1 = 2$ , pero los valores reales que toma Y están más alejados de la media que los valores reales que toma X.

La noción de varianza expresa matemáticamente qué tan lejos de la media es probable que estén los valores de una variable aleatoria. La varianza de una variable aleatoria X con media  $E \langle EX \rangle$  es

$$\begin{aligned}
 \text{Var} \langle EX \rangle &= E \langle (X - E \langle EX \rangle)^2 \rangle \\
 &= E \langle X^2 \rangle - 2E \langle X \rangle E \langle X \rangle + E^2 \langle X \rangle \\
 &= E \langle X^2 \rangle - 2E^2 \langle X \rangle + E^2 \langle X \rangle \\
 &= E \langle X^2 \rangle - E^2 \langle X \rangle ; \tag{C.27}
 \end{aligned}$$

Para justificar la igualdad  $E \langle E(X - E \langle X \rangle)^2 \rangle = E \langle X^2 \rangle - E^2 \langle X \rangle$ , tenga en cuenta que debido a que  $E \langle EX \rangle$  es un número real y no una variable aleatoria, también lo es  $E^2 \langle X \rangle$ . La igualdad  $E \langle E(X - E \langle X \rangle)^2 \rangle = E \langle X^2 \rangle - E^2 \langle X \rangle$

se sigue de la ecuación (C.22), con una DE  $\sigma^2_X$ . Reescribiendo la ecuación (C.27) se obtiene una expresión para la expectativa del cuadrado de una variable aleatoria:

$$E(X^2) = D(\sigma^2_X) + E^2(X) \quad (C.28)$$

La varianza de una variable aleatoria  $X$  y la varianza de  $aX$  están relacionadas (ver Ejercicio C.3-10):

$$\text{Var}(aX) = a^2 \text{Var}(X)$$

Cuando  $X$  e  $Y$  son variables aleatorias independientes,

$$\text{Var}(XY) = \text{Var}(X)\text{Var}(Y)$$

En general, si  $n$  variables aleatorias  $X_1; X_2; \dots; X_n$  son independientes por pares, entonces

$$\text{Var}(\sum_{i=1}^n X_i) = \sum_{i=1}^n \text{Var}(X_i) \quad (C.29)$$

La desviación estándar de una variable aleatoria  $X$  es la raíz cuadrada no negativa de la varianza de  $X$ . La desviación estándar de una variable aleatoria  $X$  a veces se denota como  $\sigma_X$  o simplemente cuando la variable aleatoria  $X$  se entiende por contexto. Con esta notación, la varianza de  $X$  se denota como  $\sigma^2_X$ .

### Ejercicios

#### C.3-1

Supongamos que lanzamos dos dados ordinarios de 6 caras. ¿Cuál es la expectativa de la suma de los dos valores que muestra? ¿Cuál es la expectativa del máximo de los dos valores que muestra?

#### C.3-2

Un arreglo  $A \in \mathbb{R}^{1 \times n}$  contiene  $n$  números distintos que están ordenados al azar, siendo cada permutación de los  $n$  números igualmente probable. ¿Cuál es la expectativa del índice del elemento máximo en la matriz? ¿Cuál es la expectativa del índice del elemento mínimo en la matriz?

#### C.3-3

Un juego de carnaval consiste en tres dados en una jaula. Un jugador puede apostar un dólar a cualquiera de los números del 1 al 6. Se agita la jaula y el pago es el siguiente. Si el número del jugador no aparece en ninguno de los dados, pierde su dólar. De lo contrario, si su número aparece exactamente en  $k$  de los tres dados, para  $k \in \{1, 2, 3\}$ , se queda con su dólar y gana  $k$  dólares más. ¿Cuál es su ganancia esperada al jugar el juego de carnaval una vez?

C.3-4

Argumente que si  $X$  e  $Y$  son variables aleatorias no negativas, entonces

$E \max(X; Y) \geq E(X) + E(Y)$ :

C.3-5 ?

Sean  $X$  e  $Y$  variables aleatorias independientes. Demuestre que  $f_X(x) f_Y(y)$  son independientes para cualquier elección de funciones  $f$  y  $g$ .

C.3-6 ?

Sea  $X$  una variable aleatoria no negativa y suponga que  $E(X)$  está bien definida.

Demostrar la desigualdad de Markov:

$$\Pr(X \geq t) \leq \frac{E(X)}{t} \quad (C.30)$$

para todo  $t > 0$ .

C.3-7 ?

Sea  $S$  un espacio muestral, y sean  $X$  y  $X_0$  variables aleatorias tales que  $X_s = X_0$  para todo  $s \in S$ . Demuestre que para cualquier constante real  $t$ ,

$\Pr(X \geq t) \leq \Pr(X_0 \geq t)$ :

C.3-8

¿Qué es mayor: la esperanza del cuadrado de una variable aleatoria o el cuadrado de su esperanza?

C.3-9

Demuestre que para cualquier variable aleatoria  $X$  que solo tome los valores 0 y 1, tenemos  $\text{Var}(X) \leq E(X)(1 - E(X))$ .

C.3-10

Demuestre que  $\text{Var}(X) \leq E(X)(1 - E(X))$  a partir de la definición (C.27) de varianza.

## C.4 Las distribuciones geométrica y binomial

Podemos pensar en el lanzamiento de una moneda como un caso de prueba de Bernoulli, que es un experimento con solo dos resultados posibles: éxito, que ocurre con probabilidad  $p$ , y fracaso, que ocurre con probabilidad  $q = 1 - p$ . Cuando hablamos de ensayos de Bernoulli en conjunto, queremos decir que los ensayos son mutuamente independientes y, a menos que digamos específicamente lo contrario, que cada uno tiene la misma probabilidad  $p$  de éxito. Dos

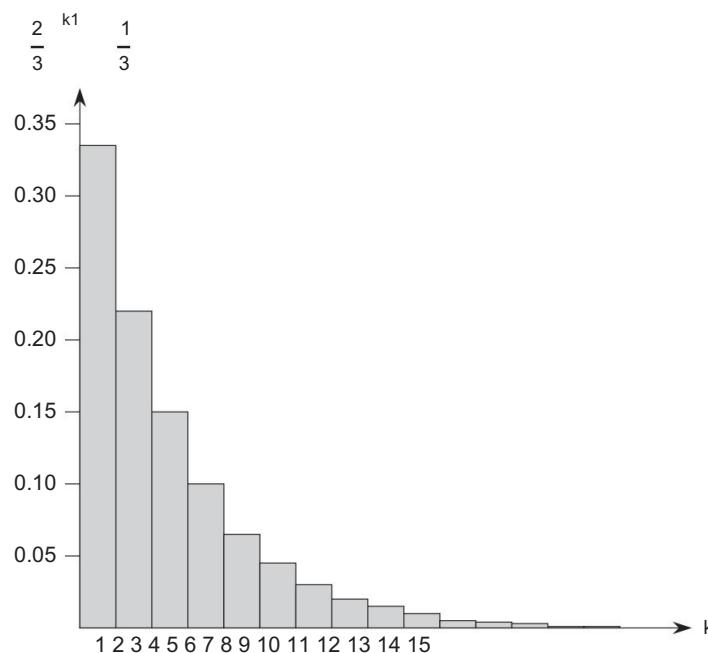


Figura C.1 Una distribución geométrica con probabilidad  $p = 1/3$  de éxito y una probabilidad  $q = 1 - p$  de fracaso. La expectativa de la distribución es  $E(X) = 3$ .

importantes distribuciones surgen de los ensayos de Bernoulli: la distribución geométrica y la distribución binomial.

#### La distribución geométrica

Suponga que tenemos una secuencia de ensayos de Bernoulli, cada uno con una probabilidad  $p$  de éxito y una probabilidad  $q = 1 - p$  de fracaso. ¿Cuántas pruebas ocurren antes de que obtengamos un éxito? Definamos la variable aleatoria  $X$  como el número de intentos necesarios para obtener un éxito. Entonces  $X$  tiene valores en el rango  $\{1; 2; \dots; g\}$ , y para  $k \geq 1$ ,

$$P(X=k) = q^{k-1}p; \quad (C.31)$$

ya que tenemos  $k-1$  fracasos antes del acierto. Se dice que una distribución de probabilidad que satisface la ecuación (C.31) es una distribución geométrica. La Figura C.1 ilustra tal distribución.

Suponiendo que  $q < 1$ , podemos calcular la expectativa de una distribución geométrica usando identidad (A.8):

$$\begin{aligned}
 E \text{ EX D } X_1 &= \frac{k q k_1 p}{k D_1} \\
 &\stackrel{D}{=} \frac{\frac{p}{q} X_1}{\frac{k}{k D_0}} \frac{k q k}{q} \\
 &\stackrel{D}{=} \frac{\frac{p}{q}}{\frac{q}{q}} \frac{1}{\frac{q}{q}} \frac{q}{q} \\
 &\stackrel{D}{=} \frac{q p}{q} = \frac{p}{p^2} \\
 D_1 &= p
 \end{aligned} \tag{C.32}$$

Por lo tanto, en promedio, toma  $1 = p$  intentos antes de que obtengamos un éxito, un resultado intuitivo. La varianza, que se puede calcular de manera similar, pero usando el Ejercicio A.1-3, es

$$\text{Var EX D } q=p^2 : \tag{C.33}$$

Como ejemplo, supongamos que lanzamos repetidamente dos dados hasta obtener un siete o un once. De los 36 resultados posibles, 6 dan un siete y 2 dan un once. Así, la probabilidad de éxito es  $p = 8/36 = 2/9$ , y debemos tirar  $1 = p = 2/9$  veces en promedio para obtener un siete o un once.

### La distribución binomial

¿Cuántos éxitos ocurren durante  $n$  pruebas de Bernoulli, donde ocurre un éxito con probabilidad  $p$  y un fracaso con probabilidad  $q = 1 - p$ ? Defina la variable aleatoria  $X$  como el número de éxitos en  $n$  intentos. Entonces  $X$  tiene valores en el rango  $f_0; 1; \dots; n$ , y para  $k = 0; 1; \dots; n$ ,

$$\Pr f_X D k = \frac{n!}{k!(n-k)!} p^k q^{n-k} ; \tag{C.34}$$

puesto que hay  $\binom{n}{k}$  formas de elegir cuáles  $k$  de los  $n$  intentos son exitosos, y la probabilidad de que cada uno ocurra es  $p^k q^{n-k}$ . Una distribución de probabilidad que satisface la ecuación (C.34) se dice que es una distribución binomial. Por conveniencia, definimos la familia de distribuciones binomiales usando la notación

$$b.k! n; P/D \quad k! p^k (1-p)^{n-k} : \tag{C.35}$$

La figura C.2 ilustra una distribución binomial. El nombre "binomial" proviene del lado derecho de la ecuación (C.34) siendo el  $k$ -ésimo término de la expansión de  $(p+q)^n$ . En consecuencia, dado que  $p = q = 1/2$ ,

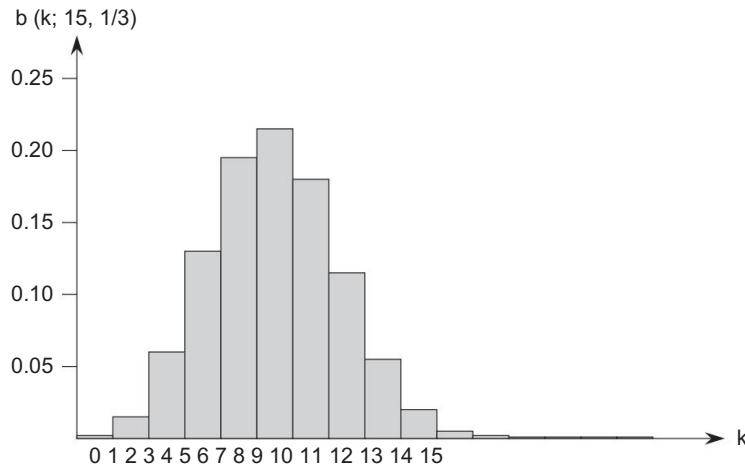


Figura C.2 La distribución binomial  $b(k; 15, 1/3)$  resultante de  $n = 15$  ensayos de Bernoulli, cada uno con probabilidad  $p = 1/3$  de éxito. La expectativa de la distribución es  $np = 5$ .

$$\mathbb{E}[X_n] = \sum_{k=0}^{n-p} k \cdot \Pr[X_n = k] \quad (C.36)$$

como requiere el axioma 2 de los axiomas de probabilidad.

Podemos calcular la expectativa de una variable aleatoria que tiene una distribución binomial a partir de las ecuaciones (C.8) y (C.36). Sea  $X$  una variable aleatoria que sigue la distribución binomial  $b(k; n, p)$ , y sea  $q = 1 - p$ . Por la definición de expectativa, tenemos

$$\mathbb{E}[X] = \sum_{k=0}^n k \Pr[X = k]$$

$$\mathbb{E}[X] = \sum_{k=0}^n k \cdot b(k; n, p)$$

$$\mathbb{E}[X] = \sum_{k=1}^n k \frac{n!}{k!(n-k)!} p^k q^{n-k}$$

$$\mathbb{E}[X] = np \sum_{k=1}^n \frac{n!}{k!(n-k)!} p^k q^{n-k} \quad (\text{por ecuación (C.8)})$$

$$\mathbb{E}[X] = np \sum_{k=0}^n \frac{n!}{k!(n-k)!} p^k q^{n-k}$$

$$\begin{aligned} D \text{ npXn1} &= b.kl n 1; \text{ pag/} \\ D np &\quad \stackrel{kD0}{=} \quad (\text{por ecuación (C.36)) .}) \end{aligned} \quad (C.37)$$

Al usar la linealidad de la expectativa, podemos obtener el mismo resultado con sustancialmente menos álgebra. Sea  $X_i$  la variable aleatoria que describe el número de éxitos en la  $i$ -ésima prueba. Entonces  $E \sum X_i = D p 1 C q 0 D p$ , y por la linealidad de la expectativa (ecuación (C.21)), el número esperado de éxitos para  $n$  intentos es

$$\begin{aligned} E \sum X_i &= \sum E X_i \\ &= \sum np_i \\ &= np: \end{aligned} \quad (C.38)$$

Podemos usar el mismo enfoque para calcular la varianza de la distribución. Ecuación (C.27), tenemos  $\text{Var } \sum X_i = \sum \text{Var } X_i$ . Usando  $E^2 X_i - E X_i^2$ . Dado que  $X_i$  solo tiene valores 0 y 1, tenemos  $E^2 X_i = E X_i$ . Usando la  $D X_i = p$ , lo que implica  $E X_i^2 = E X_i D p$ . Por eso,

$$\text{Var } \sum X_i = \sum np_i (1-p) : \quad (C.39)$$

Para calcular la varianza de  $X$ , aprovechamos la independencia de los  $n$  ensayos; así, por la ecuación (C.29),

$$\begin{aligned} \text{Var } \sum X_i &= \sum \text{Var } X_i \\ &= \sum np_i (1-p) \\ &= npq: \end{aligned} \quad (C.40)$$

Como muestra la Figura C.2, la distribución binomial  $b.kl n; p/$  aumenta con  $k$  hasta que alcanza la media  $np$ , y luego disminuye. Podemos probar que la distribución siempre se comporta de esta manera observando la razón de términos sucesivos:

$$\begin{aligned}
 & \frac{b.kl n; p/}{\text{fondo } 1l n; pag/} \stackrel{D}{=} \frac{\frac{k}{n} pkqnk}{\frac{n}{k} k! \underset{\text{norte}}{pk1qnkC1}} \\
 & \quad \stackrel{D}{=} \frac{\frac{n}{k} k! \underset{\text{norte}}{pk1qnkC1}}{\frac{n}{k} \underset{\text{norte}}{nS.k 1/S.nk C 1/S.p kS.nk}} \\
 & \quad \stackrel{D}{=} \frac{\frac{n}{k} \underset{\text{norte}}{nS.k 1/S.nk C 1/S.p kS.nk}}{\frac{n}{k} \underset{\text{norte}}{S.nS.q .nk C 1/p}} \\
 & \quad \stackrel{D}{=} \frac{\frac{n}{k} \underset{\text{norte}}{S.nS.q .nk C 1/p}}{\frac{n}{k} \underset{\text{norte}}{kq .n C 1/pk D}} \\
 & \quad \stackrel{D}{=} \frac{\frac{n}{k} \underset{\text{norte}}{kq .n C 1/pk D}}{1} \\
 & \quad \stackrel{D}{=} \frac{\frac{n}{k} \underset{\text{norte}}{kq .n C 1/pk D}}{C kq} \tag{C.41}
 \end{aligned}$$

Esta relación es mayor que 1 precisamente cuando  $.n C 1/pk$  es positivo. En consecuencia,  $b.kl n; p/ > bk 1l n; p/$  para  $k < .n C 1/p$  (la distribución aumenta), y  $b.kl n; p/ < \text{fondo } 1l n; p/$  para  $k > .n C 1/p$  (la distribución disminuye).

Si  $k D .n C 1/p$  es un número entero, entonces  $b.kl n; p/ \leq \text{fondo } 1l n; p/$ , por lo que la distribución tiene entonces dos máximos: en  $k D .n C 1/p$  y en  $k+1 D .n C 1/p$ .

De lo contrario, alcanza un máximo en el único entero  $k$  que se encuentra en el rango  $np q < k < .n C 1/p$ .

El siguiente lema proporciona un límite superior en la distribución binomial.

### Lema C.1

Sea  $n > 0$ , sea  $0 < p < 1$ , sea  $q = 1 - p$ , y sea  $0 < k < n$ . Entonces

$$b.kl n; pag/ \leq \frac{np}{k} \frac{k}{n} \frac{nq}{nk} \leq \frac{nk}{nk}$$

Prueba Usando la ecuación (C.6), tenemos

$$\begin{aligned}
 b.kl n; P/D &= \frac{k!}{n!} \frac{(np)^k}{(1-p)^{n-k}} \frac{(nq)^{n-k}}{(1-q)^k} \\
 &\stackrel{\text{norte}}{=} \frac{k!}{n!} \frac{(np)^k}{(1-p)^{n-k}} \frac{(nq)^{n-k}}{(1-q)^k} \frac{(kn)!}{(nk)!(nk)!} \frac{pkqnk}{pkqnk} \\
 &\stackrel{D}{=} \frac{np}{k} \frac{k}{n} \frac{nq}{nk} \leq \frac{nk}{nk}
 \end{aligned}$$

### Ejercicios

#### C.4-1

Verifique el axioma 2 de los axiomas de probabilidad para la distribución geométrica.

#### C.4-2

¿Cuántas veces en promedio debemos lanzar 6 monedas justas antes de obtener 3 caras y 3 cruces?

## C.4-3

Muestre que  $b.kl n; p/ D bn kl n; q/$ , donde  $q = 1 - p$ .

## C.4-4

Muestre que valor del máximo de la distribución binomial  $b.kl n; p/$  es aproximadamente  $1 = p^2 npq$ , donde  $q = 1 - p$ .

## C.4-5 ?

Muestre que la probabilidad de ningún éxito en  $n$  ensayos de Bernoulli, cada uno con probabilidad  $p$   $D = 1/n$ , es aproximadamente  $1 = e^{-n}$ . Muestre que la probabilidad de exactamente un éxito también es aproximadamente  $1 = e^{-n}$ .

## C.4-6 ?

El profesor Rosencrantz lanza una moneda justa  $n$  veces, al igual que el profesor Guildenstern. Demuestra que la probabilidad de que obtengan el mismo número de caras es  $\frac{2^n}{4^n}$ . (Sugerencia: para el profesor Rosencrantz, llame a la cabeza un éxito; para el profesor Guildenstern, llame a la cruz un éxito). Use su argumento para verificar la identidad

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

## C.4-7 ?

Demuestre que para  $0 \leq k \leq n$ ,

$$b.kl n; 1 = 2 / 2^n H_k = n/n$$

donde  $H_x$  es la función de entropía (C.7).

## C.4-8 ?

Considere  $n$  ensayos de Bernoulli, donde para  $i = 1, 2, \dots, n$ , el ensayo  $i$ -ésimo tiene probabilidad  $p_i$  de éxito, y sea  $X$  la variable aleatoria que denota el número total de éxitos. Sea  $p_i$  para todo  $i = 1, 2, \dots, n$ . Demostrar que para  $0 \leq k \leq n$ ,

$$\Pr\{X \leq k\} = \sum_{i=0}^{k-1} b.i.l n; p_i /$$

## C.4-9 ?

Sea  $X$  la variable aleatoria para el número total de éxitos en un conjunto  $A$  de  $n$

Pruebas de Bernoulli, donde la  $i$ -ésima prueba tiene una probabilidad  $p_i$  de éxito, y sea  $X_0$  la variable aleatoria para el número total de éxitos en un segundo conjunto  $A_0$  de  $n$  pruebas de Bernoulli, donde la  $i$ -ésima prueba tiene una probabilidad  $p_0$  de éxito. Demostrar que para  $0 \leq k \leq n$ ,

$\Pr fX \geq k$   $\Pr fX = k$  :

(Sugerencia: muestre cómo obtener los ensayos de Bernoulli en A0 mediante un experimento que involucre los ensayos de A y use el resultado del ejercicio C.3-7).

## ? C.5 Las colas de la distribución binomial

La probabilidad de tener al menos, o como máximo, k éxitos en n ensayos de Bernoulli, cada uno con una probabilidad p de éxito, a menudo tiene más interés que la probabilidad de tener exactamente k éxitos. En esta sección, investigamos las colas de la distribución binomial: las dos regiones de la distribución  $b.kl n; p/$  que están lejos de la media  $np$ . Demostraremos varias cotas importantes en (la suma de todos los términos en) una cola.

Primero proporcionamos un límite en la cola derecha de la distribución  $b.kl n; p/$ . Podemos determinar los límites en la cola izquierda invirtiendo los roles de éxitos y fracasos.

### Teorema C.2

Considere una secuencia de n ensayos de Bernoulli, donde el éxito ocurre con probabilidad p. Sea X la variable aleatoria que denota el número total de éxitos. Entonces, para  $0 \leq k \leq n$ , la probabilidad de al menos k éxitos es

$$\Pr fX \geq k \leq D X_n \quad b.il n; p/$$

no sé

norte

k! paquete :

Prueba para  $S f1; 2; : : : ; ng$ , dejamos que AS denote el evento de que la i-ésima prueba es un éxito para cada  $i \leq S$ . Claramente,  $\Pr fAS g D pk si jSj D k$ . Tenemos

$$\Pr fX \geq k \leq D \Pr f\text{existe } S f1; 2; : : : ; ng W jSj D k y ASg$$

$$D PR [ \quad \text{COMO}$$

$$Sf1;2;:::;ngWjSjDk$$

$$X \quad pr fasg$$

(por desigualdad (C.19))

$$Sf1;2;:::;ngWjSjDk$$

D

k! paquete :

El siguiente corolario reafirma el teorema de la cola izquierda de la distribución binomial. En general, le dejaremos a usted adaptar las demostraciones de una cola a la otra.

### Corolario C.3

Considere una secuencia de  $n$  pruebas de Bernoulli, donde el éxito ocurre con probabilidad  $p$ . Si  $X$  es la variable aleatoria que denota el número total de éxitos, entonces para  $0 \leq k \leq n$ , la probabilidad de como máximo  $k$  éxitos es

$$\Pr_{\text{D}} f(X \leq k) = \sum_{i=0}^k \frac{n!}{i!(n-i)!} p^i q^{n-i}$$

■

Nuestro próximo límite se refiere a la cola izquierda de la distribución binomial. su corolario muestra que, lejos de la media, la cola izquierda disminuye exponencialmente.

### Teorema C.4

Considere una secuencia de  $n$  pruebas de Bernoulli, donde el éxito ocurre con probabilidad  $p$  y el fracaso con probabilidad  $q = 1 - p$ . Sea  $X$  la variable aleatoria que denota el número total de éxitos. Entonces, para  $0 < k < np$ , la probabilidad de menos de  $k$  éxitos es

$$\Pr_{\text{D}} f(X < k) = \sum_{i=0}^{k-1} \frac{n!}{i!(n-i)!} p^i q^{n-i} < \frac{kq}{np - kq}$$

Prueba Acotamos la serie  $\sum_{i=0}^{k-1} \frac{n!}{i!(n-i)!} p^i q^{n-i}$  por una serie geométrica usando la técnica de la Sección A.2, página 1151. Para  $i = 1, 2, \dots, k$ , tenemos de la ecuación (C.41),  $\frac{p}{q} < 1$ :

$$\begin{aligned} & \frac{p}{q} < \frac{\frac{p}{q} + 1}{1 - \frac{p}{q}} \\ & \frac{p}{q} < \frac{1}{1 - \frac{p}{q}} \end{aligned}$$

si dejamos

$$\begin{aligned} & \frac{kq}{np/p} \\ & < \frac{kq}{np/p} \\ & \frac{nq}{k} \\ & \frac{np}{k} \\ & < 1; \end{aligned}$$

resulta que

$$b_i l n; p / < x b_i l n; pag /$$

para  $0 < i < k$ . Aplicando iterativamente esta desigualdad  $k_i$  veces, obtenemos

$$b_i l n; p / < x_{k_i} b_i l n; pag /$$

para  $0 < i < k$ , y por lo tanto

$$k_1 \quad k_1$$

$$\sum_{i=0}^k b_i l n; p / < \sum_{i=0}^{k_1} x_{k_i} b_i l n; pag /$$

$$\begin{aligned} & < b_i l n; p / X_1 \quad x_i \\ & \quad i=0 \\ & \frac{x}{x q} b_i l n; p / 1 \\ & \frac{b_i l n; p / : np}{k} \end{aligned}$$

■

### Corolario C.5

Considere una secuencia de  $n$  pruebas de Bernoulli, donde el éxito ocurre con probabilidad  $p$  y el fracaso con probabilidad  $q = 1 - p$ . Entonces, para  $0 < k < np = 2$ , la probabilidad de menos de  $k$  éxitos es menor de la mitad de la probabilidad de menos de  $k + 1$  éxitos.

Prueba Como  $k = np = 2$ , tenemos

$$\frac{kq}{np} \quad \frac{.np=2/q}{np} \quad np = 2/q$$

$$\frac{np=2/q}{np=2} \quad 1 \quad (C.42)$$

ya que  $q < 1$ . Sea  $X$  la variable aleatoria que denota el número de éxitos, el teorema C.4 y la desigualdad (C.42) implican que la probabilidad de obtener menos de  $k$  éxitos es

$$\Pr_{\text{ID0}} fX < kg \leq b \cdot il n; p / < b \cdot kl n; pag / :$$

Así tenemos

$$\begin{aligned} \frac{\Pr fX < kg}{\Pr fX < k C 1g} &\stackrel{D}{=} \frac{P_{k+1} b \cdot il n; pag /}{P_{k+1} b \cdot il n; p /} \\ &\stackrel{\text{Paquetes}}{=} \frac{P_{k+1} b \cdot il n; p /}{P_{k+1} b \cdot il n; p / C b \cdot kl n; pag /} \\ &< 1 = 2 ; \end{aligned}$$

desde  $P_{k+1} b \cdot il n; p / < b \cdot kl n; pag /$ . ■

Los límites en la cola derecha siguen de manera similar. El ejercicio C.5-2 le pide que los pruebe.

#### Corolario C.6

Considere una secuencia de  $n$  intentos de Bernoulli, donde el éxito ocurre con probabilidad  $p$ . Sea  $X$  la variable aleatoria que denota el número total de éxitos. Entonces, para  $np < k < n$ , la probabilidad de más de  $k$  éxitos es

$$\Pr fX > kg \leq D Xn \stackrel{iDkC1}{=} \frac{b \cdot il n; pag /}{n; p / : k np} < \frac{nk/p b \cdot kl}{n; p / : k np} \quad ■$$

#### Corolario C.7

Considere una secuencia de  $n$  ensayos de Bernoulli, donde el éxito ocurre con probabilidad  $p$  y el fracaso con probabilidad  $q = 1 - p$ . Entonces, para  $np < k < n$ , la probabilidad de más de  $k$  éxitos es menor de la mitad de la probabilidad de más de  $k - 1$  éxitos. ■

El siguiente teorema considera  $n$  intentos de Bernoulli, cada uno con una probabilidad  $p_i$  de éxito, para  $i = 1, 2, \dots, n$ . Como muestra el corolario subsiguiente, podemos usar el

teorema para proporcionar un límite en la cola derecha de la distribución binomial estableciendo  $\pi_D$  para cada prueba.

### Teorema C.8

Considere una secuencia de  $n$  pruebas de Bernoulli, donde en la  $i$ -ésima prueba, para  $i \in \{1, 2, \dots, n\}$ , el éxito ocurre con probabilidad  $\pi_i$  y el fracaso ocurre con probabilidad  $q_i = 1 - \pi_i$ . Sea  $X$  la variable aleatoria que describe el número total de éxitos y sea  $E(X)$ .

Entonces para  $r >$ ,

$$\Pr[f_X \geq r] \leq \frac{\binom{n}{r} \pi_r^r (1-\pi_r)^{n-r}}{r!}$$

Demostración Dado que para cualquier  $\epsilon > 0$ , la función  $e_\epsilon x$  es estrictamente

$$e_\epsilon x / \epsilon \text{ creciente en } x, \Pr[f_X \geq r] \leq e_\epsilon r \Pr[f_X \geq r] \quad (\text{C.43})$$

donde determinaremos  $\epsilon$  más adelante. Usando la desigualdad de Markov

$$(C.30), \text{ obtenemos } \Pr[f_X \geq r] \leq e_\epsilon r E(e_\epsilon X) / e_\epsilon r \quad (\text{C.44})$$

El grueso de la prueba consiste en acotar  $E(e_\epsilon X)$  y sustituir  $\epsilon$  por un valor adecuado en la desigualdad (C.44). Primero, evaluamos  $E(e_\epsilon X)$ . Usando la técnica de indicadores de variables aleatorias (ver Sección 5.2), sea  $X_i$  el resultado del  $i$ -ésimo ensayo de Bernoulli es un éxito para  $i \in \{1, 2, \dots, n\}$ ; es decir,  $X_i$  es la variable aleatoria que es 1 si el  $i$ -ésimo ensayo de Bernoulli es un éxito y 0 si es un fracaso. De este modo,

$$E(X) = \sum_{i=1}^n E(X_i) = \sum_{i=1}^n p_i$$

y por la linealidad de la expectativa,

$$E(e_\epsilon X) = E(e_\epsilon \sum_{i=1}^n X_i) = \sum_{i=1}^n E(e_\epsilon X_i) = \sum_{i=1}^n p_i$$

lo que implica

$$E(X) = \sum_{i=1}^n p_i$$

Para evaluar  $E(e_\epsilon X)$ , sustituimos por  $X = \sum_{i=1}^n X_i$ , obteniendo

$$E(e_\epsilon X) = E(e_\epsilon \sum_{i=1}^n X_i) = \sum_{i=1}^n E(e_\epsilon X_i) = \sum_{i=1}^n p_i$$

$$E(X) = \sum_{i=1}^n p_i$$

$$E(X) = \sum_{i=1}^n p_i$$

lo cual se sigue de (C.24), ya que la independencia mutua de las variables aleatorias  $X_i$  implica la independencia mutua de las variables aleatorias  $e_i X_i p_i /$  (ver Ejercicio C.3-5). Por la definición de

$$\begin{aligned} \text{expectativa, } E \sum_{i=1}^n e_i X_i p_i / &= \sum_{i=1}^n e_i p_i \\ &\leq \sum_{i=1}^n e_i \\ &= \exp(\sum_{i=1}^n e_i) ; \end{aligned} \tag{C.45}$$

donde  $\exp(x)$  denota la función exponencial:  $\exp(x) = e^x$ . (La desigualdad (C.45) se sigue de las desigualdades  $e_i \geq 0$ ,  $p_i \leq 1$ ,  $e_i p_i \leq e_i$  y  $e_i p_i \leq 1$ , y la última línea se sigue de la desigualdad (3.12).) En consecuencia,

$$\begin{aligned} E e_i X_i p_i / &= \sum_{i=1}^n e_i p_i \\ &\leq \exp(\sum_{i=1}^n e_i) \\ &= \exp(\sum_{i=1}^n e_i) \pi e_i ! \\ &\leq \exp(\sum_{i=1}^n e_i) \pi e_i ! \end{aligned} \tag{C.46}$$

desde  $\pi e_i = \prod_{j \neq i} p_j$ . Por tanto, de la ecuación (C.43) y de las desigualdades (C.44) y (C.46), se sigue que

$$\Pr\{X_n \geq r\} \leq \exp(-r \ln p) : \tag{C.47}$$

Eligiendo  $r = n \ln p$  (vea el Ejercicio C.5-7), obtenemos

$$\begin{aligned} \Pr\{X_n \geq r\} &\leq \exp(-n \ln p) \\ &= \exp(n \ln(1-p)) \\ &= \frac{1}{\sqrt[n]{1-p}} \end{aligned}$$

■

Cuando se aplica a los ensayos de Bernoulli en los que cada ensayo tiene la misma probabilidad de éxito, el teorema C.8 produce el siguiente corolario que limita la cola derecha de una distribución binomial.

## Corolario C.9

Considere una secuencia de  $n$  pruebas de Bernoulli, donde en cada prueba ocurre el éxito con probabilidad  $p$  y el fracaso ocurre con probabilidad  $q = 1 - p$ . Entonces para  $r > np$ ,

$$\Pr f X \geq np \geq D X_n \quad \text{b.kl n; pag/}$$

$$\frac{\text{kdnpcrc}}{\text{npe}} \quad \frac{r}{r}$$

Demostración Por la ecuación (C.37), tenemos DE  $\sum_{i=0}^n \Pr f X_i = np$ . ■

## Ejercicios

C.5-1 ?

¿Qué es menos probable: no obtener cara cuando lanza una moneda normal  $n$  veces o obtener menos de  $n$  cara cuando lanza la moneda  $4n$  veces?

C.5-2 ?

Demostar los Corolarios C.6 y C.7.

C.5-3 ?

Muestra esa

$$\Pr_{\substack{i=0 \\ i \neq 0}} \left( \sum_{i=0}^n \Pr f X_i < .a \right) \leq \Pr_{\substack{i=0 \\ i \neq 0}} \left( \sum_{i=0}^n \Pr f X_i < \frac{n}{2} \right) \leq \Pr_{\substack{i=0 \\ i \neq 0}} \left( \sum_{i=0}^n \Pr f X_i < \frac{n}{2} + \frac{1}{2} \right) = \Pr_{\substack{i=0 \\ i \neq 0}} \left( \sum_{i=0}^n \Pr f X_i = \frac{n}{2} \right)$$

para todo  $a > 0$  y todo  $k$  tal que  $0 < k < na = a \sqrt{n}$ .

C.5-4 ?

Demuestre que si  $0 < k < np$ , donde  $0 < p < 1$  y  $q = 1 - p$ , entonces

$$\Pr_{\substack{i=0 \\ i \neq 0}} \left( \sum_{i=0}^n \Pr f X_i < np \right) \leq \Pr_{\substack{i=0 \\ i \neq 0}} \left( \sum_{i=0}^n \Pr f X_i < \frac{np}{k} \right) \leq \Pr_{\substack{i=0 \\ i \neq 0}} \left( \sum_{i=0}^n \Pr f X_i < \frac{np}{k} + \frac{1}{k} \right) = \Pr_{\substack{i=0 \\ i \neq 0}} \left( \sum_{i=0}^n \Pr f X_i = \frac{np}{k} \right)$$

C.5-5 ?

Demuestre que las condiciones del teorema C.8 implican que

$$\Pr f X \geq r \quad \frac{\text{norte}}{\text{r}}$$

De manera similar, demuestre que las condiciones del Corolario C.9 implican que

$$\Pr f np \leq X \leq r \quad \frac{\text{norte}}{\text{r}}$$

C.5-6 ?

Considere una secuencia de  $n$  pruebas de Bernoulli, donde en la  $i$ -ésima prueba, para  $i \in D$ :  
 $2; \dots; n$ , el éxito ocurre con probabilidad  $p_i$  y el fracaso ocurre con probabilidad  $q_i = 1 - p_i$ .

Sea  $X$  la variable aleatoria que describe el número total de éxitos y sea  $D$  el espacio de éxitos. Demostrar que para  $r$

$$0,$$

$$\Pr\{X = r\} = \frac{r!}{n!} \prod_{i=1}^n p_i^{e_i} q_i^{1-e_i}$$

(Sugerencia: Demuestre que  $\pi_i q_i \leq C q_i e_i^{1/2} \leq 2$ . Luego siga el esquema de la prueba del Teorema C.8, usando esta desigualdad en lugar de la desigualdad (C.45).)

C.5-7 ?

Demuestre que elegir  $r \in D$  para minimizar el lado derecho de la desigualdad (C.47).

## Problemas

### C-1 Pelotas y recipientes

En este problema, investigamos el efecto de varias suposiciones sobre el número de formas de colocar  $n$  pelotas en  $b$  recipientes distintos.

- a. Suponga que las  $n$  bolas son distintas y que su orden dentro de un contenedor no importa.  
Argumenta que el número de formas de colocar las bolas en los contenedores es  $b^n$ .
- b. Suponga que las bolas son distintas y que las bolas en cada recipiente están ordenadas.  
Demuestre que hay exactamente  $\binom{n+1}{b-1}$  maneras de colocar las bolas en los contenedores. (Sugerencia: considere el número de formas de colocar  $n$  bolas distintas y  $b-1$  espacios indistinguibles en una fila).
- c. Suponga que las bolas son idénticas y, por lo tanto, su orden dentro de un contenedor no importa. Demuestre que el número de formas de colocar las bolas en los contenedores es  $\binom{n+b-1}{b-1}$  (Sugerencia: De los arreglos en la parte (b), ¿cuántos se repiten si las bolas se hacen idénticas?).
- d. Suponga que las bolas son idénticas y que ningún recipiente puede contener más de una bola, de modo que  $n \leq b$ . Demuestra que el número de formas de colocar las bolas es  $\binom{n}{b}$ .
- e. Suponga que las bolas son idénticas y que no se puede dejar ningún recipiente vacío. Suponiendo que  $n \geq b$ , demuestre que el número de formas de colocar las bolas es  $\binom{n-1}{b-1}$ .

### Notas del apéndice

Los primeros métodos generales para resolver problemas de probabilidad se discutieron en una famosa correspondencia entre B. Pascal y P. de Fermat, que comenzó en 1654, y en un libro de C. Huygens en 1657. La teoría rigurosa de la probabilidad comenzó con el trabajo de J. Bernoulli en 1713 y A. De Moivre en 1730. P.-S. Laplace, S.-D. Poisson y C.F. Gauss.

Las sumas de variables aleatorias fueron estudiadas originalmente por P.L. Chebyshev y A.A. Markov. A.N. Kolmogorov axiomatizó la teoría de la probabilidad en 1933. Chernoff [66] y Hoeffding [173] proporcionaron límites en las colas de las distribuciones. El trabajo seminal en estructuras combinatorias aleatorias fue realizado por P. Erdős.

Knuth [209] y Liu [237] son buenas referencias para la combinatoria elemental y el conteo. Los libros de texto estándar como Billingsley [46], Chung [67], Drake [95], Feller [104] y Rozanov [300] ofrecen introducciones completas a la probabilidad.

## D Matrices

Las matrices surgen en numerosas aplicaciones, incluida, entre otras, la computación científica. Si ha visto matrices anteriormente, gran parte del material de este apéndice le resultará familiar, pero parte de él podría ser nuevo. La Sección D.1 cubre las definiciones y operaciones básicas de matrices, y la Sección D.2 presenta algunas propiedades básicas de matrices.

---

### D.1 Matrices y operaciones con matrices

En esta sección, revisamos algunos conceptos básicos de la teoría de matrices y algunas propiedades fundamentales de las matrices.

#### Matrices y vectores

Una matriz es un arreglo rectangular de números. Por ejemplo,

ANUNCIO	a <sub>11</sub>	a <sub>12</sub>	a <sub>13</sub>								
	a <sub>21</sub>	a <sub>22</sub>	a <sub>23</sub>								
	123	456									
	0										

(D.1)

es una matriz de 2 3 AD .a<sub>ij</sub> /, donde para i D 1; 2 y j D 1; 2; 3, denotamos el elemento de la matriz en la fila i y la columna j por a<sub>ij</sub> . Usamos letras mayúsculas para denotar matrices y letras minúsculas subíndice correspondientes para denotar sus elementos. Denotamos el conjunto de todas las mn matrices con entradas de valor real por R<sub>mn</sub> y, en general, el conjunto de mn matrices con entradas extraídas de un conjunto S por S<sub>mn</sub>.

La traspuesta de una matriz A es la matriz A<sup>T</sup> obtenida al intercambiar las filas y columnas de A. Para la matriz A de la ecuación (D.1),

1 4  
en D    2 5  
            3 6

Un vector es un arreglo unidimensional de números. Por ejemplo,

2  
xD    3  
      5

es un vector de tamaño 3. A veces llamamos vector  $n$  a un vector de longitud  $n$ . Usamos letras minúsculas para denotar vectores, y denotamos el  $i$ -ésimo elemento de un vector de tamaño  $n \times 1$  por  $x_i$ , para  $i \in \{1; 2; \dots; n\}$ . Tomamos la forma estándar de un vector como un vector columna equivalente a una matriz  $n \times 1$ ; el vector fila correspondiente se obtiene

tomando la

transpuesta:  $x^T \in \mathbb{R}^{1 \times 3}$ . El vector unitario  $e_i$  es el vector cuyo  $i$ -ésimo elemento es 1 y todos los demás elementos son 0. Por lo general, el tamaño de un vector unitario es claro por el contexto.

Una matriz cero es una matriz cuyas entradas son 0. Dicha matriz a menudo se denota como 0, ya que la ambigüedad entre el número 0 y una matriz de 0 suele resolverse fácilmente a partir del contexto. Si se pretende una matriz de 0, entonces el tamaño de la matriz también debe derivarse del contexto.

### Matrices cuadradas

Las matrices cuadradas  $n \times n$  surgen con frecuencia. Varios casos especiales de matrices cuadradas son de particular interés: 1.

Una matriz diagonal tiene  $a_{ij} = 0$  siempre que  $i \neq j$ . Debido a que todos los elementos fuera de la diagonal son cero, podemos especificar la matriz enumerando los elementos a lo largo de la diagonal:

$$\begin{matrix} & & & & 0 \\ & 0 & a_{22} & \cdots & 0 \\ & \vdots & \vdots & \ddots & \vdots \\ \text{diag.} & a_{11}; a_{22}; \cdots; a_{nn} & 0 & 0 & \cdots; a_{nn} \end{matrix}$$

2. La matriz identidad de  $n \times n$   $I_n$  es una matriz diagonal con 1 a lo largo de la diagonal:

En D diag.1; 1; : : : ; 1/

$$\begin{matrix} & & & 0 \\ & 0 & 1 & \cdots & 0 \\ D & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \cdots & 1 \end{matrix}$$

Cuando I aparece sin un subíndice, derivamos su tamaño del contexto. La  $i$ -ésima columna de una matriz identidad es el vector unitario  $e_i$ .

3. Una matriz tridiagonal T es aquella para la cual  $t_{ij} = 0$  si  $j \neq i$ . Las entradas distintas de cero aparecen solo en la diagonal principal, inmediatamente arriba de la diagonal principal ( $t_{i,i+1}$  para  $i = 1, 2, \dots, n-1$ ), o inmediatamente debajo de la diagonal principal ( $t_{i,i-1}$  para  $i = 1, 2, \dots, n-1$ ):

4. Una matriz triangular superior U es aquella para la cual  $u_{ij} = 0$  si  $i > j$ .  
Todas las entradas debajo de la diagonal son cero:

Una matriz triangular superior es triangular superior unitaria si tiene todos los 1 a lo largo de la diagonal.

5. Una matriz triangular inferior L es aquella para la cual  $l_{ij} = 0$  si  $i < j$ .  
encima de la diagonal son cero:

```

LD    l21 l22 :: 0
      :: ln1 ln2 :: ln3
l11 0

```

Una matriz triangular inferior es una unidad triangular inferior si tiene todos los 1 a lo largo de la diagonal.

6. Una matriz de permutación P tiene exactamente un 1 en cada fila o columna y ceros en cualquier otro lugar. Un ejemplo de una matriz de permutación es

$$\begin{array}{l} \text{PD} \\ \quad \begin{array}{r} 00010 \\ 10000 \\ 00001 \\ 00100 \\ \hline 01000 \end{array} \end{array}$$

Tal matriz se llama matriz de permutación porque multiplicar un vector  $x$  por una matriz de permutación tiene el efecto de permutar (reorganizar) los elementos de  $x$ . El ejercicio D.1-4 explora propiedades adicionales de las matrices de permutación.

7. Una matriz simétrica A satisface la condición  $AD = AT$ . Por ejemplo,

$$\begin{array}{r} 123 \\ 264 \ 345 \end{array}$$

es una matriz simétrica.

#### Operaciones básicas de matriz

Los elementos de una matriz o vector son números de un sistema numérico, como los números reales, los números complejos o los números enteros módulo primo. El sistema numérico define cómo sumar y multiplicar números. Podemos ampliar estas definiciones para abarcar la suma y la multiplicación de matrices.

Definimos la suma de matrices de la siguiente manera. Si  $A$  y  $B$  son matrices  $m \times n$ , entonces su matriz suma  $C$  es la matriz  $m \times n$  definida por

$$c_{ij} = a_{ij} + b_{ij}$$

para  $i = 1, 2, \dots, m$ ;  $j = 1, 2, \dots, n$ . Es decir, la suma de matrices se realiza por componentes. Una matriz cero es la identidad para la suma de matrices:

$$A + 0 = 0 + A = A$$

Si  $\alpha$  es un número y  $A$  es una matriz, entonces  $\alpha A$  es el múltiplo escalar de  $A$  que se obtiene al multiplicar cada uno de sus elementos por  $\alpha$ . Como caso especial, definimos el negativo de una matriz  $A$  como  $-A$ , de modo que la  $ij$ -ésima entrada de  $-A$  es  $-\alpha_{ij}$ . De este modo,

$$A + (-A) = (-A) + A = 0$$

Usamos el negativo de una matriz para definir la resta de matrices:  $ABDAC \cdot B/$ .

Definimos la multiplicación de matrices de la siguiente manera. Empezamos con dos matrices A y B que son compatibles en el sentido de que el número de columnas de A es igual al número de filas de B. (En general, siempre se supone que una expresión que contiene un producto de matriz AB implica que las matrices A y B son compatibles.) Si  $AD \cdot aik/$  es una matriz  $mn$  y  $BD \cdot bkj/$  es una matriz  $np$ , entonces su producto de matriz  $CD AB$  es la matriz  $mp$   $CD \cdot cij /$ , donde

$$cij D Xn_{kD1} aikbkj \quad (D.2)$$

para  $i \leq 1; 2; \dots; m$  y  $j \leq 1; 2; \dots; p$ . El procedimiento MULTIPLICACIÓN DE MATRICES CUADRADAS de la Sección 4.2 implementa la multiplicación de matrices de manera directa en función de la ecuación (D.2), suponiendo que las matrices son cuadradas:  $m \leq n \leq p$ . Para multiplicar  $nn$  matrices, SQUARE-MATRIX-MULTIPLY realiza  $n^3$  multiplicaciones y  $n^2 \cdot n^1$  sumas, por lo que su tiempo de ejecución es  $\sim n^3$ .

Las matrices tienen muchas (pero no todas) de las propiedades algebraicas típicas de los números.

Las matrices de identidad son identidades para la multiplicación de matrices:

$$ImA D Aln DA$$

para cualquier matriz  $mn$  A. Multiplicar por una matriz cero da como resultado una matriz cero:

$$A 0 D 0$$

La multiplicación de matrices es asociativa:

$$A.BC / D .AB/C$$

para matrices compatibles A, B y C. La multiplicación de matrices se distribuye sobre la suma:

$$AB CC / D AB C AC .BCC / DD BD :$$

$$C CD :$$

Para  $n > 1$ , la multiplicación de  $nn$  matrices no es commutativa. Por ejemplo, si

0 1 0 ANUNCIO 0	y BD	0 0 1 0 , entonces
-----------------------	------	-----------------------

AB D	1 0
	0 0

y

MALO	0 0
	0 1

Definimos productos matriz-vector o productos vector-vector como si el vector fuera la matriz equivalente n 1 (o una matriz 1 n, en el caso de un vector fila). Por lo tanto, si A es una matriz mn y x es un vector n, entonces Ax es un vector m. Si x e y son n-vectores, entonces

$$x^T y \equiv \sum_{i=1}^n x_i y_i$$

es un número (en realidad, una matriz de 1 1) llamado producto interno de x e y. La matriz xy<sup>T</sup> es una matriz Z de nn denominada producto exterior de xey, con  $Z_{ij} \equiv x_i y_j$ . La norma (euclíadiana) kxk de un n-vector x está definida por

$$\|x\|^2 \equiv \sum_{i=1}^n x_i^2 \quad \text{y} \quad \|x\| = \sqrt{\sum_{i=1}^n x_i^2}$$

Por lo tanto, la norma de x es su longitud en el espacio euclíadiano de n dimensiones.

### Ejercicios

#### D.1-1

Muestre que si A y B son matrices nn simétricas, entonces también lo son ACB y A B.

#### D.1-2

Demuestre que .AB/T D BTAT y que ATA es siempre una matriz simétrica.

#### D.1-3

Demuestre que el producto de dos matrices triangulares inferiores es triangular inferior.

#### D.1-4

Demuestre que si P es una matriz de permutación de nn y A es una matriz de nn, entonces el producto de matrices PA es A con sus filas permutadas y el producto de matrices AP es A con sus columnas permutadas. Demuestre que el producto de dos matrices de permutación es una matriz de permutación.

## D.2 Propiedades básicas de la matriz

En esta sección, definimos algunas propiedades básicas relacionadas con las matrices: inversas, dependencia e independencia lineal, rango y determinantes. También definimos la clase de matrices definidas positivas.

### Matriz inversa, rangos y determinantes

Definimos la inversa de una matriz A de nn como la matriz de nn, denominada A<sup>-1</sup> (si existe), tal que AA<sup>-1</sup> = I = A<sup>-1</sup>A. Por ejemplo,

$$\begin{matrix} 1 & 1 & & 1 \\ & & 0 & \\ 1 & 0 & & \end{matrix} \quad \begin{matrix} & & 1 \\ & & 0 \\ & 1 & & 1 \\ & & & 1 \end{matrix}$$

Muchas matrices nn distintas de cero no tienen inversas. Una matriz sin inversa se llama no invertible o singular. Un ejemplo de una matriz singular distinta de cero es

$$\begin{matrix} 1 & 0 \\ 1 & 0 \end{matrix} :$$

Si una matriz tiene una inversa, se llama invertible o no singular. Las matrices inversas, cuando existen, son únicas. (Consulte el ejercicio D.2-1.) Si A y B son matrices nn no singulares, entonces

$$BA^{-1} = A^{-1}B$$

La operación inversa commuta con la operación de transposición:

$$A^{-1}T = T^{-1}A$$

Los vectores  $x_1, x_2, \dots, x_n$  son linealmente dependientes si existen coeficientes  $c_1, c_2, \dots, c_n$ , no todos los cuales son cero, tales que  $c_1x_1 + c_2x_2 + \dots + c_nx_n = 0$ .

Los vectores fila  $x_1 = [1, 2, 3]$ ,  $x_2 = [2, 6, 4]$ ,  $x_3 = [4, 1, 9]$  son linealmente dependientes, por ejemplo, ya que  $2x_1 + 3x_2 = 2x_3$ . Si los vectores no son linealmente dependientes, son linealmente independientes. Por ejemplo, las columnas de una matriz identidad son linealmente independientes.

El rango de columna de una matriz A mn distinta de cero es el tamaño del conjunto más grande de columnas linealmente independientes de A. De manera similar, el rango de fila de A es el tamaño del conjunto más grande de filas linealmente independientes de A. Una propiedad fundamental de cualquier matriz A es que su rango de fila siempre es igual al rango de su columna, por lo que simplemente podemos referirnos al rango de A. El rango de una matriz mn es un número entero entre 0 y  $\min(m, n)$ , inclusive. (El rango de una matriz cero es 0, y el rango de una matriz identidad nn es n). Una definición alternativa, pero equivalente y a menudo más útil, es que el rango de una matriz A distinta de cero mn es el número más pequeño r tal que existen matrices B y C de tamaños respectivos mr y rn tales que  $AC = B$ :

Una matriz cuadrada de nn tiene rango completo si su rango es n. Una matriz mn tiene rango de columna completo si su rango es n. El siguiente teorema da una propiedad fundamental de los rangos.

**Teorema D.1**

Una matriz cuadrada tiene rango completo si y sólo si es no singular. ■

Un vector nulo para una matriz A es un vector x distinto de cero tal que  $Ax = 0$ . El siguiente teorema (cuya demostración se deja como ejercicio D.2-7) y su corolario relacionan las nociones de rango de columna y singularidad con vectores nulos.

**Teorema D.2**

Una matriz A tiene rango de columna completo si y solo si no tiene un vector nulo. ■

**Corolario D.3**

Una matriz cuadrada A es singular si y sólo si tiene un vector nulo. ■

El ij-ésimo menor de una matriz A de  $n \times n$ , para  $n > 1$ , es la matriz  $\begin{smallmatrix} . & . & . \\ . & . & . \\ . & . & . \end{smallmatrix}$  que se obtiene eliminando la i-ésima fila y la j-ésima columna de A. Definimos el determinante de una matriz A de  $n \times n$  recursivamente en términos de sus menores por

$$\text{si } n = 1 : \quad \det A = a_{11}$$

$$\text{Xn } \frac{1}{n!} C_{ij} a_{ij} \det A \quad / \quad \text{si } n > 1 : \\ \det A = \sum_{j=1}^n (-1)^{i+j} a_{ij} \det A_{ij}$$

El término  $\frac{1}{n!} C_{ij} a_{ij} \det A_{ij}$  es conocido como el cofactor del elemento  $a_{ij}$ .

Los siguientes teoremas, cuyas demostraciones se omiten aquí, expresan propiedades fundamentales del determinante.

**Teorema D.4 (Propiedades determinantes)**

El determinante de una matriz cuadrada A tiene las siguientes propiedades:

Si cualquier fila o columna de A es cero, entonces  $\det A = 0$ .

El determinante de A se multiplica por si todas las entradas de cualquier fila (o cualquier columna) de A se multiplican por .

El determinante de A no cambia si las entradas de un renglón (respectivamente, la columna) se suman a las de otro renglón (respectivamente, la columna).

El determinante de A es igual al determinante de  $A^T$ .

El determinante de A se multiplica por 1 si se intercambian dos filas (o dos columnas).

Además, para cualquier matriz cuadrada A y B, tenemos  $\det(AB) = \det A \det B$ . ■

**Teorema D.5**

Una matriz A de nn es singular si y sólo si  $\det A = 0$ . ■

**Matrices definidas positivas**

Las matrices definidas positivas juegan un papel importante en muchas aplicaciones. Una matriz A de nn es definida positiva si  $x^T A x > 0$  para todos los n-vectores  $x \neq 0$ . Por ejemplo, la matriz identidad es definida positiva, ya que para cualquier vector distinto de cero  $x \in \mathbb{R}^n$ ,  $x^T I_n x > 0$ .

$$\begin{aligned} x^T I_n x &= x_1^2 + x_2^2 + \dots + x_n^2 \\ &\geq 0: \end{aligned}$$

Las matrices que surgen en las aplicaciones suelen ser definidas positivas debido al siguiente teorema.

**Teorema D.6**

Para cualquier matriz A con rango de columna completo, la matriz ATA es definida positiva.

Prueba Debemos demostrar que  $x^T A^T A x > 0$  para cualquier vector x distinto de cero. Para cualquier vector x,

$$x^T A^T A x = \|Ax\|^2 \quad (\text{por el ejercicio D.1-2})$$

$$= \|Ax\|^2$$

Note que  $\|Ax\|^2$  es simplemente la suma de los cuadrados de los elementos del vector Ax.

Por lo tanto,  $\|Ax\|^2 \geq 0$ . Si  $\|Ax\|^2 = 0$ , todo elemento de Ax es 0, es decir

$Ax = 0$ . Dado que A tiene rango de columna completo,  $Ax = 0$  implica  $x = 0$ , por el teorema D.2.

Por lo tanto, ATA es definido positivo. ■

La sección 28.3 explora otras propiedades de las matrices definidas positivas.

**Ejercicios****D.2-1**

Demuestre que las matrices inversas son únicas, es decir, si B y C son inversas de A, entonces  $B = C$ .

**D.2-2**

Demostrar que el determinante de una matriz triangular inferior o triangular superior es igual al producto de sus elementos diagonales. Demuestre que la inversa de una matriz triangular inferior, si existe, es triangular inferior.

D.2-3

Demuestre que si  $P$  es una matriz de permutación, entonces  $P$  es invertible, su inversa es  $P^T$  y  $P^T$  es una matriz de permutación.

D.2-4

Sean  $A$  y  $B$  nn matrices tales que  $AB = I$ .  
Demostrar que si se obtiene  $A_0$  de  $A$  sumando la fila  $j$  a la fila  $i$ , luego restando la columna  $i$  de la columna  $j$  de  $B$  se obtiene el inverso  $B_0$  de  $A_0$ .

D.2-5

Sea  $A$  una matriz nn no singular con elementos complejos. Demostrar que toda entrada de  $A_1$  es real si y sólo si toda entrada de  $A$  es real.

D.2-6

Demuestre que si  $A$  es una matriz nn no singular, simétrica, entonces  $A_1$  es simétrica.  
Demuestre que si  $B$  es una matriz mn arbitraria, entonces la matriz mm dada por el producto  $BAB^T$  es simétrica.

D.2-7

Demostrar el teorema D.2. Es decir, demuestre que una matriz  $A$  tiene rango de columna completo si y solo si  $Ax = 0$  implica  $x = 0$ . (Sugerencia: exprese la dependencia lineal de una columna con respecto a las demás como una ecuación matricial-vectorial).

D.2-8

Demuestre que para cualesquiera dos matrices compatibles  $A$  y  $B$ ,

$$\text{rango}(AB) \leq \min(\text{rango}(A), \text{rango}(B)) ;$$

donde la igualdad se cumple si  $A$  o  $B$  es una matriz cuadrada no singular. (Sugerencia: use la definición alternativa del rango de una matriz).

## Problemas

Matriz de Vandermonde D-1

Números dados  $x_0; x_1; \dots; x_n$ , demuestre que el determinante de la matriz de Vandermonde

$$\begin{vmatrix}
 & x_0^2 & x_0^n \\
 V_{(x_0; x_1; \dots; x_n)} & \vdots & \vdots \\
 & x_1^2 & x_1^n \\
 & \vdots & \vdots \\
 & x_n^2 & x_n^n
 \end{vmatrix}$$

es

$$\det(V \cdot x_0; x_1; \dots; x_n) / D Y \quad \cdot x_k x_j / : 0j \\ < kn_1$$

(Sugerencia: multiplique la columna  $i$  por  $x_0$  y súmela a la columna  $i$   $C 1$  para  $i \in D \cap \{1; n-2, \dots, 1\}$ , y luego use la inducción).

D-2 Permutaciones definidas por multiplicación matriz-vector sobre GF.2/ Una clase de permutaciones de los enteros en el conjunto  $S_n$   $D = \{0; 1; 2; \dots; 2^n - 1\}$  se define por multiplicación de matrices sobre GF.2/. Para cada entero  $x$  en  $S_n$ , vemos su representación binaria como un vector de  $n$  bits

$$\begin{matrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{matrix}$$

donde  $x \in D \cap \{0, 1\}^n$ . Si  $A$  es una matriz  $n \times n$  en la que cada entrada es 0 o 1, entonces podemos definir una permutación que asigna cada valor  $x$  en  $S_n$  al número cuya representación binaria es el producto matriz-vector  $Ax$ . Aquí, realizamos toda la aritmética sobre GF.2/: todos los valores son 0 o 1 y, con una excepción, se aplican las reglas habituales de suma y multiplicación. La excepción es que  $1 \cdot C 1 = 0$ . Puede pensar que la aritmética sobre GF.2/ es como la aritmética de enteros regular, excepto que usa solo el bit menos significativo.

Como ejemplo, para  $S_2$   $D = \{0; 1; 2; 3\}$ , la matriz

$$\begin{matrix} 1 & 0 \\ \text{ANUNCIO} & 1 & 1 \end{matrix}$$

define la siguiente permutación  $A$ :  $A[0] = D[0]$ ,  $A[1] = D[3]$ ,  $A[2] = D[2]$ ,  $A[3] = D[1]$ . Para ver por qué  $A[3] = D[1]$ , observe que, trabajando en GF.2/,

$$\begin{array}{rcc} A[3]/D & 1 & 0 & 1 \\ & 1 & 1 & 1 \\ \hline & 1 & 1 & C 0 & 1 \\ & 1 & 1 & C & 1 & 1 \\ \hline & 1 & & & & \\ & 0 & & \vdots & & \end{array}$$

que es la representación binaria de 1.

Para el resto de este problema, trabajamos sobre GF.2/ $\mathbb{Z}$ , y todas las entradas de matriz y vector son 0 o 1. Definimos el rango de una matriz 0-1 (una matriz para la cual cada entrada es 0 o 1) sobre GF.2/ $\mathbb{Z}$  igual que para una matriz regular, pero con toda la aritmética que determina la independencia lineal realizada sobre GF.2/ $\mathbb{Z}$ . Definimos el rango de una matriz A nn 0-1 por

$\text{RA} = \{Ax \mid x \in S_n\}$  ;

de manera que  $\text{RA}$  es el conjunto de números en  $S_n$  que podemos producir al multiplicar cada valor  $x$  en  $S_n$  por A.

- Si  $r$  es el rango de la matriz A, demuestre que  $\text{jR.A}/j \leq r$ . Concluya que A define una permutación en  $S_n$  sólo si A tiene rango completo.

Para una matriz A dada de nn y un valor dado y  $\in \text{RA}$ , definimos la preimagen de y por

$P_A(y) = \{x \mid Ax = y\}$  ;

para que  $P_A(y)$  es el conjunto de valores en  $S_n$  que corresponde a y cuando se multiplica por A.

- Si  $r$  es el rango de la matriz A de nn y  $y \in \text{RA}$ , demuestre que  $\text{jP}_A(y)/j \leq nr$

Sea 0 mn, y supongamos que dividimos el conjunto  $S_n$  en bloques de números consecutivos, donde el i-ésimo bloque consta de los  $2m$  números  $i2m; i2m + 1; i2m + 2; \dots; i2m + m - 1$ . Para cualquier subconjunto S de  $S_n$ , defina BS; m/ sea el conjunto de bloques de tamaño  $2m$  de  $S_n$  que contienen algún elemento de S. Por ejemplo, cuando n = 3, m = 1 y SD f1; 4; 5 g, luego BS; m/ consta de los bloques 0 (puesto que el 1 está en el bloque 0) y 2 (dados que tanto el 4 como el 5 están en el bloque 2).

- Sea r el rango de la submatriz inferior izquierda .nm/m de A, es decir, la matriz formada al tomar la intersección de las nm filas inferiores y las m columnas más a la izquierda de A. Sea S cualquier bloque de tamaño  $2m$  de  $S_n$ , y sea S0 D fy W y D Ax para alguna x  $\in S$ . Demuestre que  $\text{jB.S0}/j \leq m/r$  y eso para cada bloque en B.S0 ; m/, exactamente  $2mr$  números en S se asignan a ese bloque.

Debido a que multiplicar el vector cero por cualquier matriz produce un vector cero, el conjunto de permutaciones de  $S_n$  definido al multiplicar por nn matrices 0-1 con rango completo sobre GF.2/ $\mathbb{Z}$  no puede incluir todas las permutaciones de  $S_n$ . Extendamos la clase de permutaciones definidas por la multiplicación matriz-vector para incluir un término aditivo, de modo que  $x \in S_n$  se corresponda con  $Ax + C$ , donde C es un vector de n bits y la suma se realiza sobre GF.2/ $\mathbb{Z}$ . por ejemplo, cuando

1 0	
ANUNCIO	
1 1	

y

$$\begin{matrix} & 0 \\ cd & \vdots \\ & 1 \end{matrix}$$

obtenemos la siguiente permutación  $A;c: A;c.0/ D 2, A;c.1/ D 1, A;c.2/ D 0, A;c.3/ D 3$ . Llamamos a cualquier permutación que asigna  $x \in S_n$  a  $Ax \in C^m$ , para algunos  $n \times n$  matriz  $A$  con rango completo y algún vector  $c$  de  $n$  bits, una permutación lineal.

d. Use un argumento de conteo para mostrar que el número de permutaciones lineales de  $S_n$  es mucho menor que el número de permutaciones de  $S_n$ .

mi. Dé un ejemplo de un valor de  $n$  y una permutación de  $S_n$  que no se pueda lograr mediante ninguna permutación lineal. (Sugerencia: para una permutación dada, piense en cómo la multiplicación de una matriz por un vector unitario se relaciona con las columnas de la matriz).

---

### Notas del apéndice

Los libros de texto de álgebra lineal proporcionan mucha información básica sobre las matrices. Los libros de Strang [323, 324] son particularmente buenos.



# Bibliografía

- [1] Milton Abramowitz e Irene A. Stegun, editores. Manual de funciones matemáticas. Dover, 1965.
- [2] GM Adel'son-Vel'skiĭ y EM Landis. Un algoritmo para la organización de la información. Matemáticas soviéticas Doklady, 3 (5): 1259–1263, 1962.
- [3] Alok Aggarwal y Jeffrey Scott Vitter. La complejidad de entrada/salida de clasificación y relacionados problemas. Comunicaciones de la ACM, 31(9):1116–1127, 1988.
- [4] Manindra Agrawal, Neeraj Kayal y Nitin Saxena. PRIMES está en P. Annals of Mathematics, 160(2):781–793, 2004.
- [5] Alfred V. Aho, John E. Hopcroft y Jeffrey D. Ullman. El Diseño y Análisis de Algoritmos informáticos. Addison-Wesley, 1974.
- [6] Alfred V. Aho, John E. Hopcroft y Jeffrey D. Ullman. Estructuras de datos y algoritmos. Addison-Wesley, 1983.
- [7] Ravindra K. Ahuja, Thomas L. Magnanti y James B. Orlin. Flujos de red: teoría, Algoritmos y Aplicaciones. Prentice Hall, 1993.
- [8] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin y Robert E. Tarjan. Algoritmos más rápidos para el problema del camino más corto. Revista de la ACM, 37:213–223, 1990.
- [9] Ravindra K. Ahuja y James B. Orlin. Un algoritmo rápido y sencillo para el caudal máximo problema. Investigación de operaciones, 37(5):748–759, 1989.
- [10] Ravindra K. Ahuja, James B. Orlin y Robert E. Tarjan. Límites de tiempo mejorados para el problema de flujo máximo. SIAM Journal on Computing, 18(5):939–954, 1989.
- [11] Mikl'os Ajtai, Nimrod Megiddo y Orli Waarts. Algoritmos y análisis mejorados para problemas de secretaria y generalizaciones. En Actas del 36º Simposio Anual sobre Fundamentos de Ciencias de la Computación, páginas 473–482, 1995.
- [12] Selim G. Akl. El Diseño y Análisis de Algoritmos Paralelos. Prentice Hall, 1989.
- [13] Mohamad Akra y Louay Bazzi. Sobre la solución de ecuaciones lineales de recurrencia. computación Optimización y aplicaciones de la estación, 10(2):195–210, 1998.
- [14] Noga Alón. Generación de permutaciones pseudoaleatorias y algoritmos de flujo máximo. En Cartas de procesamiento de formación, 35:201–204, 1990.

- [15] Arne Anderson. Árboles de búsqueda equilibrados simplificados. En Actas del Tercer Taller sobre Algoritmos y Estructuras de Datos, volumen 709 de Lecture Notes in Computer Science, páginas 60–71. Springer, 1993.
- [16] Arne Anderson. Clasificación y búsqueda deterministas más rápidas en el espacio lineal. En Actas del 37º Simposio Anual sobre Fundamentos de Ciencias de la Computación, páginas 135–141, 1996.
- [17] Arne Andersson, Torben Hagerup, Stefan Nilsson y Rajeev Raman. ¿Ordenar en tiempo lineal? Revista de Ciencias de la Computación y Sistemas, 57: 74–93, 1998.
- [18] Tom M. Apóstol. Calculus, volumen 1. Blaisdell Publishing Company, segunda edición, 1967.
- [19] Nimar S. Arora, Robert D. Blumofe y C. Greg Plaxton. Programación de subprocessos para multiprocesadores multiprogramados. En Actas del 10º Simposio Anual de ACM sobre Algoritmos y Arquitecturas Paralelas, páginas 119–129, 1998.
- [20] Sanjeev Arora. Comprobación probabilística de demostraciones y problemas de dureza de aproximación. Tesis doctoral, Universidad de California, Berkeley, 1994.
- [21] Sanjeev Arora. La aproximabilidad de los problemas NP-difíciles. En Actas del 30º Simposio Anual de ACM sobre Teoría de la Computación, páginas 337–348, 1998.
- [22] Sanjeev Arora. Esquemas de aproximación de tiempo polinomial para el viajante de comercio euclíadiano y otros problemas geométricos. Revista de la ACM, 45(5):753–782, 1998.
- [23] Sanjeev Arora y Carsten Lund. Dureza de aproximaciones. En Dorit S. Hochbaum, editora, Algoritmos de aproximación para problemas NP-Hard, páginas 399–446. Editorial PWS, 1997.
- [24] Javed A. Aslam. Un límite simple en la altura esperada de un árbol de búsqueda binario construido aleatoriamente. Informe técnico TR2001-387, Departamento de informática de Dartmouth College, 2001.
- [25] Mikhail J. Atallah, editor. Manual de algoritmos y teoría de la computación. prensa crc, 1999.
- [26] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela y M. Protasi. Complejidad y Aproximación: Problemas de Optimización Combinatoria y sus Propiedades de Aproximación. Springer, 1999.
- [27] Shai Avidan y Ariel Shamir. Talla de costura para cambiar el tamaño de la imagen según el contenido. Trans ACM acciones sobre Gráficos, 26(3), artículo 10, 2007.
- [28] Sara Baase y Alan Van Gelder. Algoritmos Informáticos: Introducción al Diseño y Análisis. Addison-Wesley, tercera edición, 2000.
- [29] Eric Bach. Comunicación privada, 1989.
- [30] Eric Bach. Algoritmos de teoría de números. En Annual Review of Computer Science, volumen 4, páginas 119–172. Revisiones anuales, Inc., 1990.
- [31] Eric Bach y Jeffrey Shallit. Teoría de números algorítmicos: volumen I: algoritmos eficientes. Prensa del MIT, 1996.
- [32] David H. Bailey, King Lee y Horst D. Simon. Uso del algoritmo de Strassen para acelerar la solución de sistemas lineales. Revista de supercomputación, 4(4):357–371, 1990.

- [33] Surender Baswana, Ramesh Hariharan y Sandeep Sen. Algoritmos decrementales mejorados para mantener el cierre transitivo y las rutas más cortas de todos los pares. *Revista de Algoritmos*, 62(2):74–92, 2007.
- [34] R. Bayer. Árboles B binarios simétricos: estructura de datos y algoritmos de mantenimiento. *acta Informática*, 1(4):290–306, 1972.
- [35] R. Bayer y EM McCreight. Organización y mantenimiento de grandes índices ordenados. *Acta Informatica*, 1(3):173–189, 1972.
- [36] Pierre Beauchemin, Gilles Brassard, Claude Cr'epau, Claude Goutier y Carl Pomerance. La generación de números aleatorios que probablemente sean primos. *Revista de criptología*, 1(1):53–64, 1988.
- [37] Ricardo Bellman. Programación dinámica. *Prensa de la Universidad de Princeton*, 1957.
- [38] Ricardo Bellman. En un problema de enrutamiento. *Trimestral de Matemáticas Aplicadas*, 16(1):87–90, 1958.
- [39] Michael Ben-Or. Límites inferiores para árboles de cálculo algebraico. En *Actas del Decimoquinto Simposio Anual de ACM sobre Teoría de la Computación*, páginas 80–86, 1983.
- [40] Michael A. Bender, Erik D. Demaine y Martin Farach-Colton. Árboles B ajenos a la memoria caché. En *Actas del 41º Simposio Anual sobre Fundamentos de Ciencias de la Computación*, páginas 399–409, 2000.
- [41] Samuel W. Bent y John W. John. Encontrar la mediana requiere  $2n$  comparaciones. En *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, páginas 213–216, 1985.
- [42] Jon L. Bentley. Escribir programas eficientes. *Prentice Hall*, 1982.
- [43] Jon L. Bentley. Perlas de programación. *Addison-Wesley*, 1986.
- [44] Jon L. Bentley, Dorothea Haken y James B. Saxe. Un método general para resolver dividir y conquistar las recurrencias. *Noticias SIGACT*, 12(3):36–44, 1980.
- [45] Daniel Bienstock y Benjamín McClosky. Ajuste de conjuntos de enteros mixtos simplex con límites garantizados. *Optimización en línea*, julio de 2008.
- [46] Patricio Billingsley. Probabilidad y Medida. *John Wiley & Sons*, segunda edición, 1986.
- [47] Guy E. Blelloch. Escanear Primitivas y Modelos de Vectores Paralelos. Tesis de doctorado, Departamento de Ingeniería Eléctrica y Ciencias de la Computación, MIT, 1989. Disponible como Informe técnico del Laboratorio de Ciencias de la Computación del MIT MIT/LCS/TR-463.
- [48] Guy E. Blelloch. Programación de algoritmos paralelos. *Comunicaciones de la ACM*, 39(3):85–97, 1996.
- [49] Guy E. Blelloch, Phillip B. Gibbons y Yossi Matias. Programación demostrablemente eficiente para lenguajes con paralelismo de grano fino. En *Actas del 7º Simposio Anual de ACM sobre Algoritmos y Arquitecturas Paralelas*, páginas 1–12, 1995.
- [50] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest y Robert E. Tarjan. Límites de tiempo para la selección. *Revista de Ciencias de la Computación y Sistemas*, 7(4):448–461, 1973.
- [51] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall y Yuli Zhou. Cilk: un eficiente sistema de tiempo de ejecución multiproceso. *Revista de Computación Paralela y Distribuida*, 37(1):55–69, 1996.

- [52] Robert D. Blumofe y Charles E. Leiserson. Programación de cálculos de subprocesos múltiples mediante el robo de trabajo. *Revista de la ACM*, 46(5):720–748, 1999.
- [53] B'ela Bollob'as. *Gráficos aleatorios*. Prensa Académica, 1985.
- [54] Gilles Brassard y Paul Bratley. *Fundamentos de Algoritmia*. Prentice Hall, 1996.
- [55] Richard P.Brent. La evaluación paralela de expresiones aritméticas generales. *Diario de la ACM*, 21(2):201–206, 1974.
- [56] Richard P.Brent. Un algoritmo mejorado de factorización de Monte Carlo. *TBI*, 20(2):176–184, 1980.
- [57] JP Buhler, HW Lenstra, Jr. y Carl Pomerance. Factorización de números enteros con el tamiz de campo numérico. En AK Lenstra y HW Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volumen 1554 de *Lecture Notes in Mathematics*, páginas 50–94. Springer, 1993.
- [58] J. Lawrence Carter y Mark N. Wegman. Clases universales de funciones hash. *Diario de Ciencias informáticas y de sistemas*, 18(2):143–154, 1979.
- [59] Barbara Chapman, Gabriele Jost y Ruud van der Pas. *Uso de OpenMP: portátil compartido Programación de memoria en paralelo*. Prensa del MIT, 2007.
- [60] Bernardo Chazelle. Un algoritmo de árbol de expansión mínimo con tipo Ackermann inverso com complejidad. *Revista de la ACM*, 47(6):1028–1047, 2000.
- [61] Joseph Cheriyan y Torben Hagerup. Un algoritmo de flujo máximo aleatorizado. *SIAM Revista de informática*, 24(2):203–226, 1995.
- [62] Joseph Cheriyan y SN Maheshwari. Análisis de algoritmos de empuje de preflujo para un flujo de red máximo. *SIAM Journal on Computing*, 18(6):1057–1086, 1989.
- [63] Boris V. Cherkassky y Andrew V. Goldberg. Sobre la implementación del método push-relabel para el problema de flujo máximo. *Algoritmica*, 19(4):390–410, 1997.
- [64] Boris V. Cherkassky, Andrew V. Goldberg y Tomasz Radzik. Algoritmos de caminos más cortos: Teoría y evaluación experimental. *Programación matemática*, 73(2):129–174, 1996.
- [65] Boris V. Cherkassky, Andrew V. Goldberg y Craig Silverstein. Baldes, montones, listas y colas de prioridad monótonas. *SIAM Journal on Computing*, 28(4):1326–1346, 1999.
- [66] H. Chernoff. Una medida de eficiencia asintótica para pruebas de una hipótesis basada en la suma de observaciones. *Annals of Mathematical Statistics*, 23(4):493–507, 1952.
- [67] Kai Lai Chung. *Teoría de la Probabilidad Elemental con Procesos Estocásticos*. Springer, 1974.
- [68] V. Chvatal. Una heurística codiciosa para el problema de cobertura de conjuntos. *Matemáticas de Operaciones Investigación*, 4(3):233–235, 1979.
- [69] V. Chvatal. *Programación lineal*. WH Freeman and Company, 1983.
- [70] V. Chvatal, DA Klarner y DE Knuth. Problemas de investigación combinatoria seleccionados. Informe técnico STAN-CS-72-292, Departamento de informática, Universidad de Stanford, 1972.
- [71] Cilk Arts, Inc., Burlington, Massachusetts. Guía del programador de Cilk++, 2008. Disponible en <http://www.cilk.com/archive/docs/cilk1guide>.

- [72] Alan Cobham. La dificultad computacional intrínseca de las funciones. En Actas del Congreso de Lógica, Metodología y Filosofía de la Ciencia de 1964, páginas 24–30. Holanda Septentrional, 1964.
- [73] H. Cohen y HW Lenstra, Jr. Pruebas de primalidad y sumas de Jacobi. *Matemáticas de Computación*, 42(165):297–330, 1984.
- [74] D. Comer. El omnipresente árbol B. *Encuestas informáticas de ACM*, 11(2):121–137, 1979.
- [75] Stephen Cook. La complejidad de los procedimientos de demostración de teoremas. En Actas del Tercer Simposio Anual de ACM sobre Teoría de la Computación, páginas 151–158, 1971.
- [76] James W. Cooley y John W. Tukey. Un algoritmo para el cálculo automático de series complejas de Fourier. *Matemáticas de Computación*, 19(90):297–301, 1965.
- [77] Don Calderero. Modificaciones al tamiz de campos numéricos. *revista de criptología*, 6(3):169–180, 1993.
- [78] Don Coppersmith y Shmuel Winograd. Multiplicación de matrices mediante progresión aritmética. *Revista de Computación Simbólica*, 9(3):251–280, 1990.
- [79] Thomas H. Cormen, Thomas Sundquist y Leonard F. Wisniewski. Límites asintóticamente ajustados para realizar permutaciones BMMC en sistemas de discos paralelos. *SIAM Journal on Computing*, 28(1):105–136, 1998.
- [80] Don Dailey y Charles E. Leiserson. Uso de Cilk para escribir programas de ajedrez multiprocesador. En HJ van den Herik y B. Monien, editores, *Advances in Computer Games*, volumen 9, páginas 25–52. Universidad de Maastricht, Países Bajos, 2001.
- [81] Paolo D'Alberto y Alexandru Nicolau. Multiplicación de matrices de Strassen adaptativo. En *Proceedings of the 21st Annual International Conference on Supercomputing*, páginas 284–292, junio de 2007.
- [82] Sanjoy Dasgupta, Christos Papadimitriou y Umesh Vazirani. *Algoritmos*. McGraw-Hill, 2008.
- [83] Roman Dementiev, Lutz Kettner, Jens Mehnert y Peter Sanders. Ingeniería de una estructura de datos de lista ordenada para claves de 32 bits. En *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, páginas 142–151, enero de 2004.
- [84] Camil Demetrescu y Giuseppe F. Italiano. Totalmente dinámico, todos los pares de rutas más cortas con pesos de borde reales. *Revista de Ciencias de la Computación y Sistemas*, 72(5):813–837, 2006.
- [85] Eric V. Denardo y Bennett L. Fox. Métodos de ruta más corta: 1. Alcance, poda y baldes. *Investigación de operaciones*, 27(1):161–186, 1979.
- [86] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert y Robert E. Tarjan. Hashing perfecto dinámico: límites superior e inferior. *SIAM Journal on Computing*, 23(4):738–761, 1994.
- [87] Whitfield Diffie y Martin E. Hellman. Nuevas direcciones en criptografía. *transacción IEEE* ciones sobre teoría de la información, IT-22(6):644–654, 1976.
- [88] EW Dijkstra. Una nota sobre dos problemas en relación con los gráficos. *Numerische Mathematik*, 1(1):269–271, 1959.

- [89] EA Dinic. Algoritmo para la solución de un problema de caudal máximo en una red con estimación de potencia. *Matemáticas soviéticas Doklady*, 11 (5): 1277–1280, 1970.
- [90] Brandon Dixon, Monika Rauch y Robert E. Tarjan. Verificación y análisis de sensibilidad de árboles de expansión mínimos en tiempo lineal. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.
- [91] Juan D. Dixon. Pruebas de factorización y primalidad. *el mensual matemático americano*, 91(6):333–352, 1984.
- [92] Dorit Dor, Johan H'astad, Staffan Ulfberg y Uri Zwick. Sobre cotas inferiores para seleccionar la mediana. *SIAM Journal on Discrete Mathematics*, 14(3):299–311, 2001.
- [93] Dorit Dor y Uri Zwick. Selección de la mediana. *Revista SIAM sobre informática*, 28 (5): 1722–1758, 1999.
- [94] Dorit Dor y Uri Zwick. La selección de la mediana requiere comparaciones de  $0,2 C/n$ . *Revista SIAM sobre Matemáticas Discretas*, 14(3):312–325, 2001.
- [95] Alvin W. Drake. *Fundamentos de la Teoría de la Probabilidad Aplicada*. McGraw-Hill, 1967.
- [96] James R. Driscoll, Harold N. Gabow, Ruth Shrairman y Robert E. Tarjan. Montones relajados: una alternativa a los montones de Fibonacci con aplicaciones para el cálculo paralelo. *Communications of the ACM*, 31(11):1343–1354, 1988.
- [97] James R. Driscoll, Neil Sarnak, Daniel D. Sleator y Robert E. Tarjan. Hacer que las estructuras de datos sean persistentes. *Revista de Ciencias de la Computación y Sistemas*, 38(1):86–124, 1989.
- [98] Derek L. Eager, John Zahorjan y Edward D. Lazowska. Aceleración versus eficiencia en sistemas paralelos. *IEEE Transactions on Computers*, 38(3):408–423, 1989.
- [99] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volumen 10 de EATCS Monografías sobre Informática Teórica. Springer, 1987.
- [100] Jack Edmonds. Caminos, árboles y flores. *Revista canadiense de matemáticas*, 17: 449–467, 1965.
- [101] Jack Edmonds. Matroids y el algoritmo codicioso. *Programación Matemática*, 1(1):127–136, 1971.
- [102] Jack Edmonds y Richard M. Karp. Mejoras teóricas en el rendimiento algorítmico ciencia para problemas de flujo de red. *Revista de la ACM*, 19(2):248–264, 1972.
- [103] Incluso Shimón. Algoritmos de gráficos. *Prensa de informática*, 1979.
- [104] Guillermo Feller. *Una introducción a la teoría de la probabilidad y sus aplicaciones*. John Wiley & Sons, tercera edición, 1968.
- [105] Robert W. Floyd. Algoritmo 97 (RUTA MÁS CORTA). *Comunicaciones de la ACM*, 5(6):345, 1962.
- [106] Robert W. Floyd. Algoritmo 245 (TREESORT). *Comunicaciones de la ACM*, 7(12):701, 1964.
- [107] Robert W. Floyd. Permutación de información en almacenamiento idealizado de dos niveles. En Raymond E. Miller y James W. Thatcher, editores, *Complexity of Computer Computations*, páginas 105–109. Plenum Press, 1972.

- [108] Robert W. Floyd y Ronald L. Rivest. Límites de tiempo esperados para la selección. *comunicaciones de la ACM*, 18(3):165–172, 1975.
- [109] Lester R. Ford, Jr. y DR Fulkerson. *Flujos en Redes*. prensa de la universidad de princeton, 1962.
- [110] Lester R. Ford, Jr. y Selmer M. Johnson. Un problema del torneo. *El mate americano Matical Monthly*, 66(5):387–389, 1959.
- [111] Michael L. Fredman. Nuevos límites en la complejidad del problema del camino más corto. *SIAM Journal on Computing*, 5(1):83–89, 1976.
- [112] Michael L. Fredman, J'anos Koml'os y Endre Szemer'edi. Almacenar una tabla dispersa con O.1/tiempo de acceso en el peor de los casos. *Revista de la ACM*, 31(3):538–544, 1984.
- [113] Michael L. Fredman y Michael E. Saks. La complejidad de la sonda celular de las estructuras de datos dinámicos. En *Actas del vigésimo primer simposio anual de ACM sobre teoría de la computación*, páginas 345–354, 1989.
- [114] Michael L. Fredman y Robert E. Tarjan. Montones de Fibonacci y sus usos en algoritmos mejorados de optimización de redes. *Revista de la ACM*, 34(3):596–615, 1987.
- [115] Michael L. Fredman y Dan E. Willard. Superando el límite teórico de la información con árboles de fusión. *Revista de Ciencias de la Computación y Sistemas*, 47(3):424–436, 1993.
- [116] Michael L. Fredman y Dan E. Willard. Algoritmos transdicotómicos para árboles de expansión mínimos y caminos más cortos. *Revista de Ciencias de la Computación y Sistemas*, 48(3):533–551, 1994.
- [117] Matteo Frigo y Steven G. Johnson. El diseño e implementación de FFTW3. *Proceder ings del IEEE*, 93(2):216–231, 2005.
- [118] Matteo Frigo, Charles E. Leiserson y Keith H. Randall. La implementación del lenguaje multiproceso Cilk-5. En *Actas de la Conferencia ACM SIGPLAN de 1998 sobre diseño e implementación de lenguajes de programación*, páginas 212–223, 1998.
- [119] Harold N. Gabow. Búsqueda en profundidad basada en rutas para componentes fuertes y biconectados. *Cartas de procesamiento de información*, 74(3–4):107–114, 2000.
- [120] Harold N. Gabow, Z. Galil, T. Spencer y Robert E. Tarjan. Algoritmos eficientes para encontrar árboles de expansión mínimos en gráficos dirigidos y no dirigidos. *Combinatoria*, 6(2):109–122, 1986.
- [121] Harold N. Gabow y Robert E. Tarjan. Un algoritmo de tiempo lineal para un caso especial de unión de conjuntos disjuntos. *Revista de Ciencias de la Computación y Sistemas*, 30(2):209–221, 1985.
- [122] Harold N. Gabow y Robert E. Tarjan. Algoritmos de escalado más rápidos para problemas de red. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.
- [123] Zvi Galil y Oded Margalit. Todos los pares de distancias más cortas para gráficos con bordes de longitud de enteros pequeños. *Información y Computación*, 134(2):103–139, 1997.
- [124] Zvi Galil y Oded Margalit. Todos los pares de rutas más cortas para gráficos con bordes de longitud de enteros pequeños. *Revista de Ciencias de la Computación y Sistemas*, 54(2):243–254, 1997.
- [125] Parque Zvi Galil y Kunsoo. Programación dinámica con convexidad, concavidad y escasez. *Informática teórica*, 92(1):49–76, 1992.

- [126] Zvi Galil y Joel Seiferas. Coincidencia de cadenas de tiempo-espacio-óptimo. *Diario de Informática y Ciencias del sistema*, 26(3):280–294, 1983.
- [127] Igal Galperin y Ronald L. Rivest. Árboles de chivos expiatorios. En *Actas del 4º Simposio ACM-SIAM sobre algoritmos discretos*, páginas 165–174, 1993.
- [128] Michael R. Garey, RL Graham y JD Ullman. Análisis del peor de los casos de algoritmos de localización de memoria. En *Actas del Cuarto Simposio Anual de ACM sobre Teoría de la Computación*, páginas 143–150, 1972.
- [129] Michael R. Garey y David S. Johnson. *Computadoras e intratabilidad: una guía para el Teoría de la NP-Compleitud*. WH Freeman, 1979.
- [130] Saúl Gass. *Programación Lineal: Métodos y Aplicaciones*. Pub internacional Thomson publicación, cuarta edición, 1975.
- [131] Fëdorica Gavril. Algoritmos para coloración mínima, camarilla máxima, cobertura mínima por camarillas y conjunto independiente máximo de un gráfico cordal. *SIAM Journal on Computing*, 1(2):180–187, 1972.
- [132] Alan George y Joseph WH Liu. *Solución informática de gran definición positiva dispersa Sistemas*. Prentice Hall, 1981.
- [133] EN Gilbert y EF Moore. Codificaciones binarias de longitud variable. *Técnico del sistema de timbre Revista*, 38(4):933–967, 1959.
- [134] Michel X. Goemans y David P. Williamson. Algoritmos de aproximación mejorados para problemas de máximo corte y satisfacibilidad usando programación semidefinida. *Revista de la ACM*, 42(6):1115–1145, 1995.
- [135] Michel X. Goemans y David P. Williamson. El método primal-dual para algoritmos de aproximación y su aplicación a problemas de diseño de redes. En Dorit S. Hochbaum, editora, *Algoritmos de aproximación para problemas NP-Hard*, páginas 144–191. PWS Publishing Company, 1997.
- [136] Andrés V. Goldberg. Algoritmos de gráficos eficientes para computadoras secuenciales y paralelas. Tesis doctoral, Departamento de Ingeniería Eléctrica y Ciencias de la Computación, MIT, 1987.
- [137] Andrés V. Goldberg. Algoritmos de escalado para el problema de los caminos más cortos. *SIAM Journal on Computing*, 24(3):494–504, 1995.
- [138] Andrew V. Goldberg y Satish Rao. Más allá de la barrera de descomposición del flujo. *Diario de ACM*, 45(5):783–797, 1998.
- [139] Andrew V. Goldberg, Eva Tardos y Robert E. Tarjan. Algoritmos de flujo de red. En Bernhard Korte, László Lovász, Hans Jürgen Prömel y Alexander Schrijver, editores, *Paths, Flows, and VLSI-Layout*, páginas 101–164. Springer, 1990.
- [140] Andrew V. Goldberg y Robert E. Tarjan. Un nuevo enfoque para el problema de flujo máximo. *Revista de la ACM*, 35(4):921–940, 1988.
- [141] D. Goldfarb y MJ Todd. Programación lineal. En GL Nemhauser, AHG Rinnooy Kan y MJ Todd, editores, *Handbook in Operations Research and Management Science*, vol. 1, Optimización, páginas 73–170. Editores de ciencia de Elsevier, 1989.
- [142] Shafi Goldwasser y Silvio Micali. Cifrado probabilístico. *Revista de Ciencias de la Computación y Sistemas*, 28(2):270–299, 1984.

- [143] Shafi Goldwasser, Silvio Micali y Ronald L. Rivest. Un esquema de firma digital seguro contra ataques adaptativos de mensajes elegidos. *SIAM Journal on Computing*, 17(2):281–308, 1988.
- [144] Gene H. Golub y Charles F. Van Loan. *Cálculos Matriciales*. The Johns Hopkins University Press, tercera edición, 1996.
- [145] GH Gonnet. *Manual de algoritmos y estructuras de datos*. Addison-Wesley, 1984.
- [146] Rafael C. González y Richard E. Woods. *Procesando imagen digital*. Addison Wesley, 1992.
- [147] Michael T. Goodrich y Roberto Tamassia. *Estructuras de datos y algoritmos en Java*. John Wiley & Sons, 1998.
- [148] Michael T. Goodrich y Roberto Tamassia. *Diseño de algoritmos: fundamentos, análisis y Ejemplos de Internet*. John Wiley & Sons, 2001.
- [149] Ronald L. Graham. Límites para ciertas anomalías de multiprocesador. *Técnico del sistema de timbre Revista*, 45(9):1563–1581, 1966.
- [150] Ronald L. Graham. Un algoritmo eficiente para determinar el casco convexo de un conjunto plano finito. *Cartas de procesamiento de información*, 1(4):132–133, 1972.
- [151] Ronald L. Graham y Pavol Hell. Sobre la historia del problema del árbol de expansión mínima. *Anales de la Historia de la Computación*, 7(1):43–57, 1985.
- [152] Ronald L. Graham, Donald E. Knuth y Oren Patashnik. *Matemáticas Concretas*. Addison-Wesley, segunda edición, 1994.
- [153] David Gries. *La ciencia de la programación*. Springer, 1981.
- [154] M. Grötschel, László Lovász y Alexander Schrijver. *Algoritmos Geométricos y Optimización Combinatoria*. Springer, 1988.
- [155] Leo J. Guibas y Robert Sedgewick. Un marco dicromático para árboles equilibrados. En *Actas del 19º Simposio Anual sobre Fundamentos de Ciencias de la Computación*, páginas 8–21, 1978.
- [156] Dan Gusfield. *Algoritmos sobre cadenas, árboles y secuencias: informática y biología computacional*. Prensa de la Universidad de Cambridge, 1997.
- [157] H. Halberstam y RE Ingram, editores. *The Mathematical Papers of Sir William Rowan Hamilton*, volumen III (Álgebra). Prensa de la Universidad de Cambridge, 1967.
- [158] Yijie Han. Mejora de la ordenación rápida de enteros en el espacio lineal. En *Actas de la 12ª ACM Simposio SIAM sobre algoritmos discretos*, páginas 793–796, 2001.
- [159] Yijie Han. Un algoritmo  $O(n^3 \log \log n = \log n/5=4/ \text{time})$  para la ruta más corta de todos los pares. *algorítmica*, 51(4):428–434, 2008.
- [160] Frank Harary. *Teoría de grafos*. Addison-Wesley, 1969.
- [161] Gregory C. Harfst y Edward M. Reingold. Un análisis amortizado basado en el potencial de la estructura de datos union-find. *Noticias SIGACT*, 31(3):86–95, 2000.
- [162] J. Hartmanis y RE Stearns. Sobre la complejidad computacional de los algoritmos. *Transactions of the American Mathematical Society*, 117:285–306, mayo de 1965.

- [163] Michael T. Heideman, Don H. Johnson y C. Sidney Burrus. Gauss y la historia de la Transformada Rápida de Fourier. *Revista IEEE ASSP*, 1(4):14–21, 1984.
- [164] Monika R. Henzinger y Valerie King. Biconectividad totalmente dinámica y cierre transitivo. En *Actas del 36º Simposio Anual sobre Fundamentos de Ciencias de la Computación*, páginas 664–672, 1995.
- [165] Monika R. Henzinger y Valerie King. Algoritmos de gráficos totalmente dinámicos aleatorios con tiempo polilogarítmico por operación. *Revista de la ACM*, 46(4):502–516, 1999.
- [166] Monika R. Henzinger, Satish Rao y Harold N. Gabow. Cómputo de conectividad de vértices: Nuevos límites a partir de viejas técnicas. *Revista de Algoritmos*, 34(2):222–250, 2000.
- [167] Nicolás J. Higham. Explotación de la multiplicación de matriz rápida dentro del BLAS de nivel 3. *MCA Transactions on Mathematical Software*, 16(4):352–368, 1990.
- [168] W. Daniel Hillis y Jr. Guy L. Steele. Algoritmos de datos paralelos. *Comunicaciones de la ACM*, 29(12):1170–1183, 1986.
- [169] CAR Hoare. Algoritmo 63 (PARTICIÓN) y algoritmo 65 (ENCONTRAR). *Comunicaciones de la ACM*, 4(7):321–322, 1961.
- [170] CAR Hoare. Ordenación rápida. *Computer Journal*, 5(1):10–15, 1962.
- [171] Dorit S. Hochbaum. Límites eficientes para el conjunto estable, la cobertura de vértices y el problema de empaquetamiento del conjunto. *Revista Matemáticas aplicadas discretas*, 6(3):243–254, 1983.
- [172] Dorit S. Hochbaum, editora. *Algoritmos de aproximación para problemas NP-Hard*. PWS Publishing Company, 1997.
- [173] W. Hoeffding. Sobre la distribución del número de aciertos en ensayos independientes. *Anales de Estadísticas Matemáticas*, 27(3):713–721, 1956.
- [174] Micha Hofri. *Análisis Probabilístico de Algoritmos*. Springer, 1987.
- [175] Micha Hofri. *Análisis de Algoritmos*. Prensa de la Universidad de Oxford, 1995.
- [176] John E. Hopcroft y Richard M. Karp. Un algoritmo n<sup>5/2</sup> para coincidencias máximas en grafos bipartitos. *SIAM Journal on Computing*, 2(4):225–231, 1973.
- [177] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. *Introducción a la teoría de autómatas, lenguajes y computación*. Addison Wesley, tercera edición, 2006.
- [178] John E. Hopcroft y Robert E. Tarjan. Algoritmos eficientes para la manipulación de gráficos. *Comunicaciones de la ACM*, 16(6):372–378, 1973.
- [179] John E. Hopcroft y Jeffrey D. Ullman. Establecer algoritmos de fusión. *Revista SIAM en Comisión*, 2(4):294–303, 1973.
- [180] John E. Hopcroft y Jeffrey D. Ullman. *Introducción a la teoría de autómatas, lenguajes y computación*. Addison-Wesley, 1979.
- [181] Ellis Horowitz, Sartaj Sahni y Sanguthevar Rajasekaran. *Algoritmos informáticos*. Computer Science Press, 1998.
- [182] TC Hu y MT Shing. Cálculo de productos de cadenas de matrices. Parte I. *SIAM Journal on Computing*, 11(2):362–373, 1982.
- [183] TC Hu y MT Shing. Cálculo de productos de cadenas de matrices. Parte II. *SIAM Journal on Computing*, 13(2):228–251, 1984.

- [184] TC Hu y AC Tucker. Árboles de búsqueda informáticos óptimos y códigos alfabéticos de longitud variable. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.
- [185] David A. Huffman. Un método para la construcción de códigos de mínima redundancia. *Pro actas de la IRE*, 40(9):1098–1101, 1952.
- [186] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao y Thomas Turnbull. Implementación del algoritmo de Strassen para la multiplicación de matrices. En *Actas de la Conferencia ACM/IEEE de 1996 sobre supercomputación*, artículo 32, 1996.
- [187] Óscar H. Ibarra y Chul E. Kim. Algoritmos de aproximación rápida para la mochila y la suma de problemas de subconjuntos. *Revista de la ACM*, 22(4):463–468, 1975.
- [188] EJ Isaac y RC Singleton. Clasificación por cálculo de dirección. *Diario de la ACM*, 3(3):169–174, 1956.
- [189] RA Jarvis. Sobre la identificación de la envolvente convexa de un conjunto finito de puntos en el plano. *Cartas de procesamiento de información*, 2(1):18–21, 1973.
- [190] David S. Johnson. Algoritmos de aproximación para problemas combinatorios. *diario de com Computer and System Sciences*, 9(3):256–278, 1974.
- [191] David S. Johnson. La columna de NP-completitud: una guía continua: la historia del segundo probador. *Revista de Algoritmos*, 13(3):502–524, 1992.
- [192] Donald B. Johnson. Algoritmos eficientes para las rutas más cortas en redes dispersas. *Diario de ACM*, 24(1):1–13, 1977.
- [193] Richard Johnsonbaugh y Marcus Schaefer. *Algoritmos*. Pearson Prentice Hall, 2004.
- [194] A. Karatsuba y Yu. De hombre. Multiplicación de números de varios dígitos en autómatas. *Soviet Physics—Doklady*, 7(7):595–596, 1963. Traducción de un artículo en *Doklady Akademii Nauk SSSR*, 145(2), 1962.
- [195] David R. Karger, Philip N. Klein y Robert E. Tarjan. Un algoritmo aleatorio de tiempo lineal para encontrar árboles de expansión mínimos. *Revista de la ACM*, 42(2):321–328, 1995.
- [196] David R. Karger, Daphne Koller y Steven J. Phillips. Encontrar el camino oculto: Límites de tiempo para los caminos más cortos de todos los pares. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.
- [197] Howard Karloff. Programación lineal. Birkhäuser, 1991.
- [198] N. Karmarkar. Un nuevo algoritmo de tiempo polinomial para programación lineal. *combinatoria*, 4(4):373–395, 1984.
- [199] Richard M. Karp. Reducibilidad entre problemas combinatorios. En Raymond E. Miller y James W. Thatcher, editores, *Complexity of Computer Computations*, páginas 85–103. Plenum Press, 1972.
- [200] Richard M. Karp. Una introducción a los algoritmos aleatorios. *Matemática Aplicada Discreta*, 34(1–3):165–201, 1991.
- [201] Richard M. Karp y Michael O. Rabin. Algoritmos eficientes de coincidencia de patrones aleatorios. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [202] AV Karzanov. Determinación del caudal máximo en una red por el método de preflujos. *Matemáticas soviéticas Doklady*, 15 (2): 434–437, 1974.

- [203] Valerie King. Un algoritmo de verificación de árbol de expansión mínimo más simple. *algorítmica*, 18(2):263–270, 1997.
- [204] Valerie King, Satish Rao y Robert E. Tarjan. Un algoritmo de flujo máximo determinista más rápido ritmo *Revista de Algoritmos*, 17(3):447–474, 1994.
- [205] Jeffrey H. Kingston. *Algoritmos y Estructuras de Datos: Diseño, Corrección, Análisis*. Addison-Wesley, segunda edición, 1997.
- [206] DG Kirkpatrick y R. Seidel. ¿El último algoritmo de casco plano convexo? *SIAM Journal on Computing*, 15(2):287–299, 1986.
- [207] Philip N. Klein y Neal E. Young. Algoritmos de aproximación para problemas de optimización NP-hard. En *CRC Handbook on Algorithms*, páginas 34-1–34-19. Prensa CRC, 1999.
- [208] Jon Kleinberg y Eva Tardos. *Diseño de algoritmos*. Addison-Wesley, 2006.
- [209] Donald E. Knuth. Algoritmos fundamentales, volumen 1 de *The Art of Computer Programming* Addison-Wesley, 1968. Tercera edición, 1997.
- [210] Donald E. Knuth. Algoritmos semiméricos, volumen 2 de *The Art of Computer Programming* Addison-Wesley, 1969. Tercera edición, 1997.
- [211] Donald E. Knuth. Clasificación y búsqueda, volumen 3 de *El arte de la programación informática*. Addison-Wesley, 1973. Segunda edición, 1998.
- [212] Donald E. Knuth. Árboles de búsqueda binarios óptimos. *Acta Informatica*, 1(1):14–25, 1971.
- [213] Donald E. Knuth. Omicron grande y omega grande y theta grande. *Noticias SIGACT*, 8(2):18–23, 1976.
- [214] Donald E. Knuth, James H. Morris, Jr. y Vaughan R. Pratt. Coincidencia rápida de patrones en instrumentos de cuerda. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [215] J. Komlós. Verificación lineal para árboles de expansión. *Combinatoria*, 5(1):57–65, 1985.
- [216] Bernhard Korte y László Lovász. Estructuras matemáticas subyacentes a los algoritmos voraces. En F. Gecseg, editor, *Fundamentals of Computation Theory*, volumen 117 de *Lecture Notes in Computer Science*, páginas 205–209. Springer, 1981.
- [217] Bernhard Korte y László Lovász. Propiedades estructurales de greedoids. *Combinatoria*, 3(3–4):359–374, 1983.
- [218] Bernhard Korte y László Lovász. Greedoids: un marco estructural para el algoritmo voraz. En W. Pulleybank, editor, *Progress in Combinatorial Optimization*, páginas 221–243. Academic Press, 1984.
- [219] Bernhard Korte y László Lovász. Greedoids y funciones objetivo lineales. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):229–238, 1984.
- [220] Dexter C. Kozen. *El Diseño y Análisis de Algoritmos*. Springer, 1992.
- [221] David W. Krumme, George Cybenko y KN Venkataraman. Cotilleando en un tiempo mínimo. *SIAM Journal on Computing*, 21(1):111–139, 1992.
- [222] Joseph B. Kruskal, Jr. Sobre el subárbol de expansión más corto de un gráfico y el vendedor ambulante problema. *Actas de la Sociedad Matemática Estadounidense*, 7 (1): 48–50, 1956.
- [223] Leslie Lamport. Cómo hacer una computadora multiprocesador que ejecute correctamente programas multiproceso. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

- [224] Eugene L. Lawler. Optimización Combinatoria: Redes y Matroides. Holt, Rinhart, y Winston, 1976.
- [225] Eugene L. Lawler, JK Lenstra, AHG Rinnooy Kan y DB Shmoys, editores. El problema del viajante de comercio. John Wiley & Sons, 1985.
- [226] CY Lee. Un algoritmo para la conexión de caminos y sus aplicaciones. *Transacciones IRE en Computadoras electrónicas*, EC-10(3):346–365, 1961.
- [227] Tom Leighton. Límites estrechos en la complejidad de la clasificación paralela. *IEEE Transactions on Computers*, C-34(4):344–354, 1985.
- [228] Tom Leighton. Notas sobre mejores teoremas maestros para recurrencias de divide y vencerás. Apuntes de clase. Disponible en <http://citeseer.ist.psu.edu/252350.html>, octubre de 1996.
- [229] Tom Leighton y Satish Rao. Teoremas de flujo máximo y corte mínimo de múltiples productos básicos y su uso en el diseño de algoritmos de aproximación. *Revista de la ACM*, 46(6):787–832, 1999.
- [230] Daan Leijen y Judd Hall. Optimice el código administrado para máquinas de varios núcleos. *Revista MSDN*, octubre de 2007.
- [231] Debra A. Lelewel y Daniel S. Hirschberg. Compresión de datos. *Encuestas de computación ACM*, 19(3):261–296, 1987.
- [232] AK Lenstra, HW Lenstra, Jr., MS Manasse y JM Pollard. El tamiz de campo numérico. En AK Lenstra y HW Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volumen 1554 de *Lecture Notes in Mathematics*, páginas 11–42. Springer, 1993.
- [233] HW Lenstra, Jr. Factorización de enteros con curvas elípticas. *Annals of Mathematics*, 126(3):649–673, 1987.
- [234] LA Levin. Problemas de clasificación universal. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. En ruso.
- [235] Anany Levitin. Introducción al Diseño y Análisis de Algoritmos. Addison Wesley, 2007.
- [236] Harry R. Lewis y Christos H. Papadimitriou. Elementos de la Teoría de la Computación. Prentice Hall, segunda edición, 1998.
- [237] CL Liu. Introducción a las Matemáticas Combinatorias. McGraw-Hill, 1968.
- [238] L'aszl'o Lov'asz. Sobre la relación de coberturas integrales y fraccionarias óptimas. *Matemáticas discretas*, 13(4):383–390, 1975.
- [239] L'aszl'o Lov'asz y MD Plummer. Matching Theory, volumen 121 de *Annals of Discrete Mathematics*. Holanda Septentrional, 1986.
- [240] Bruce M. Maggs y Serge A. Plotkin. Árbol de expansión de costo mínimo como búsqueda de caminos problema. *Cartas de procesamiento de información*, 26(6):291–293, 1988.
- [241] Miguel principal. Estructuras de datos y otros objetos utilizando Java. Addison-Wesley, 1999.
- [242] Udi Manber. Introducción a los algoritmos: un enfoque creativo. Addison-Wesley, 1989.
- [243] Conrado Martínez y Salvador Roura. Árboles de búsqueda binarios aleatorios. *Diario de la ACM*, 45(2):288–323, 1998.
- [244] William J. Masek y Michael S. Paterson. Un algoritmo más rápido que calcula las distancias de edición de cadenas. *Revista de Ciencias de la Computación y Sistemas*, 20(1):18–31, 1980.

- [245] HA Maurer, Th. Ottmann y H.-W. Seis. Implementación de diccionarios utilizando árboles binarios de muy pequeña altura. *Cartas de procesamiento de información*, 5(1):11–14, 1976.
- [246] Ernst W. Mayr, Hans J'urgen Pr'omel y Angelika Steger, editores. *Lectures on Proof Verification and Approximation Algorithms*, volumen 1367 de *Lecture Notes in Computer Science*. Springer, 1998.
- [247] CC McGeoch. Todos los pares de caminos más cortos y el subgrafo esencial. *algorítmica*, 13(5):426–441, 1995.
- [248] MD McIlroy. Un adversario asesino para quicksort. *Software: práctica y experiencia*, 29(4):341–344, 1999.
- [249] Kurt Mehlhorn. Clasificación y búsqueda, volumen 1 de *Estructuras de datos y algoritmos*. Springer, 1984.
- [250] Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volumen 2 de *Data Structures and Algoritmos*. Springer, 1984.
- [251] Kurt Mehlhorn. Búsqueda multidimensional y geometría computacional, volumen 3 de *Estructuras de datos y algoritmos*. Springer, 1984.
- [252] Kurt Mehlhorn y Stefan N'aher. Diccionarios ordenados acotados en  $O.\log \log N / \text{time}$  y  $O(n / \text{space}$ . *Cartas de procesamiento de información*, 35(4):183–189, 1990.
- [253] Kurt Mehlhorn y Stefan N'aher. *LEDA: una plataforma para combinatoria y geometría Informática*. Prensa de la Universidad de Cambridge, 1999.
- [254] Alfred J. Menezes, Paul C. van Oorschot y Scott A. Vanstone. *Manual de Criptografía Aplicada*. Prensa CRC, 1997.
- [255] Gary L. Molinero. Hipótesis de Riemann y pruebas de primalidad. *Diario de Informática y Ciencias del sistema*, 13(3):300–317, 1976.
- [256] Juan C. Mitchell. *Fundamentos de los lenguajes de programación*. Prensa del MIT, 1996.
- [257] José SB Mitchell. Las subdivisiones de guillotina se aproximan a las subdivisiones poligonales: un esquema simple de aproximación de tiempo polinomial para TSP geométrico, k-MST y problemas relacionados. *SIAM Journal on Computing*, 28(4):1298–1309, 1999.
- [258] Luis Monier. *Algoritmos de Factorización D'Entiers*. Tesis doctoral, L'Université de Paris-Sud, 1980.
- [259] Luis Monier. Evaluación y comparación de dos pruebas de primalidad probabilística eficientes algoritmos *Informática teórica*, 12(1):97–108, 1980.
- [260] Edward F. Moore. El camino más corto a través de un laberinto. En *Actas del Simposio Internacional sobre la Teoría de la Comunicación*, páginas 285–292. Prensa de la Universidad de Harvard, 1959.
- [261] Rajeev Motwani, Joseph (Seffi) Naor y Prabhakar Raghavan. Algoritmos de aproximación aleatoria en optimización combinatoria. En Dorit Hochbaum, editora, *Algoritmos de aproximación para problemas NP-Hard*, capítulo 11, páginas 447–481. Editorial PWS, 1997.
- [262] Rajeev Motwani y Prabhakar Raghavan. *Algoritmos aleatorios*. Universidad de Cambridge Prensa, 1995.
- [263] JI Munro y V. Raman. Clasificación in situ rápida y estable con movimientos On/data. *algorítmica*, 16(2):151–160, 1996.

- [264] J. Nievergelt y EM Reingold. Árboles binarios de búsqueda de balance acotado. *SIAM Journal on Computing*, 2(1):33–43, 1973.
- [265] Ivan Niven y Herbert S. Zuckerman. *Introducción a la teoría de los números*. John Wiley & Sons, cuarta edición, 1980.
- [266] Alan V. Oppenheim y Ronald W. Schafer, con John R. Buck. *Señal de tiempo discreto Procesando*. Prentice Hall, segunda edición, 1998.
- [267] Alan V. Oppenheim y Alan S. Willsky, con S. Hamid Nawab. *Señales y Sistemas*. Prentice Hall, segunda edición, 1997.
- [268] James B. Orlin. Un algoritmo simplex de red primal de tiempo polinomial para flujos de costo mínimo. *Programación matemática*, 78(1):109–129, 1997.
- [269] Joseph O'Rourke. *Geometría computacional en C*. Cambridge University Press, segundo edición, 1998.
- [270] Christos H. Papadimitriou. *Complejidad computacional*. Addison-Wesley, 1994.
- [271] Christos H. Papadimitriou y Kenneth Steiglitz. *Optimización Combinatoria: Algoritmos y Complejidad*. Prentice Hall, 1982.
- [272] Michael S. Paterson. Avances en la selección. En *Actas de la quinta obra escandinava sobre teoría de algoritmos*, páginas 368–379, 1996.
- [273] Mihai Pătrașcu y Mikkel Thorup. Compensaciones espacio-temporales para la búsqueda de predecesores. En *Actas del 38º Simposio Anual de ACM sobre Teoría de la Computación*, páginas 232–240, 2006.
- [274] Mihai Pătrașcu y Mikkel Thorup. La aleatorización no ayuda a buscar predecesores. En *Actas del 18º Simposio ACM-SIAM sobre algoritmos discretos*, páginas 555–564, 2007.
- [275] Pavel A. Pevzner. *Biología molecular computacional: un enfoque algorítmico*. El MIT Prensa, 2000.
- [276] Steven Phillips y Jeffrey Westbrook. Equilibrio de carga en línea y flujo de red. En *Actas del 25º Simposio Anual de ACM sobre Teoría de la Computación*, páginas 402–411, 1993.
- [277] JM Pollard. Un método de Monte Carlo para la factorización. *TBI*, 15(3):331–334, 1975.
- [278] JM Pollard. Factorización con números cúbicos. En AK Lenstra y HW Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volumen 1554 de *Lecture Notes in Mathematics*, páginas 4–10. Springer, 1993.
- [279] Carlos Pomerance. Sobre la distribución de los pseudoprimos. *Matemáticas de Computación*, 37(156):587–593, 1981.
- [280] Carl Pomerance, editor. *Actas de los Simposios AMS en Matemáticas Aplicadas: Teoría Computacional de Números y Criptografía*. Sociedad Matemática Estadounidense, 1990.
- [281] William K. Pratt. *Procesando imagen digital*. John Wiley & Sons, cuarta edición, 2007.
- [282] Franco P. Preparata y Michael Ian Shamos. *Geometría computacional: una introducción*. Springer, 1985.

- [283] William H. Press, Saul A. Teukolsky, William T. Vetterling y Brian P. Flannery. Recetas numéricas en C++: El arte de la computación científica. Cambridge University Press, segunda edición, 2002.
- [284] William H. Press, Saul A. Teukolsky, William T. Vetterling y Brian P. Flannery. Recetas Numer ical: El Arte de la Computación Científica. Cambridge University Press, tercera edición, 2007.
- [285] RC Prim. Redes de conexión más cortas y algunas generalizaciones. Boletín técnico del sistema Bell, 36(6):1389–1401, 1957.
- [286] Guillermo Pugh. Saltar listas: una alternativa probabilística a los árboles equilibrados. Comunicaciones de ACM, 33(6):668–676, 1990.
- [287] Paul W. Purdom, Jr. y Cynthia A. Brown. El análisis de algoritmos. Holt, Rinhart, y Winston, 1985.
- [288] Michael O. Rabin. Algoritmos probabilísticos. En JF Traub, editor, Algorithms and Complexity: New Directions and Recent Results, páginas 21–39. Prensa Académica, 1976.
- [289] Michael O. Rabin. Algoritmo probabilístico para probar la primalidad. revista de teoría de números, 12(1):128–138, 1980.
- [290] P. Raghavan y CD Thompson. Redondeo aleatorio: una técnica para algoritmos y pruebas algorítmicas demostrablemente buenos. Combinatoria, 7(4):365–374, 1987.
- [291] Rajeev Raman. Resultados recientes sobre el problema de los caminos más cortos de fuente única. Noticias SIGACT, 28(2):81–87, 1997.
- [292] James Reinders. Intel Threading Building Blocks: equipamiento de C++ para procesadores multinúcleo Paralelismo. O'Reilly Media, Inc., 2007.
- [293] Edward M. Reingold, Jürg Nievergelt y Narsingh Deo. Algoritmos Combinatorios: Teoría y Práctica. Prentice Hall, 1977.
- [294] Edward M. Reingold, Kenneth J. Urban y David Gries. Coincidencia de cadenas KMP revisada. Cartas de procesamiento de información, 64(5):217–223, 1997.
- [295] Hans Riesel. Números primos y métodos informáticos para la factorización, volumen 126 de Progreso en Matemáticas. Birkhäuser, segunda edición, 1994.
- [296] Ronald L. Rivest, Adi Shamir y Leonard M. Adleman. Un método para obtener firmas digitales y criptosistemas de clave pública. Communications of the ACM, 21(2):120–126, 1978. Véase también la patente estadounidense 4.405.829.
- [297] Herbert Robbins. Una observación sobre la fórmula de Stirling. American Mathematical Monthly, 62(1):26–29, 1955.
- [298] DJ Rosenkrantz, RE Stearns y PM Lewis. Un análisis de varias heurísticas para la Problema del viajante de comercio. SIAM Journal on Computing, 6(3):563–581, 1977.
- [299] Salvador Roura. Un teorema maestro mejorado para las recurrencias de divide y vencerás. En Proceedings of Automata, Languages and Programming, 24th International Colloquium, ICALP'97, volumen 1256 de Lecture Notes in Computer Science, páginas 449–459. Springer, 1997.
- [300] YA Rozanov. Teoría de la probabilidad: un curso conciso. Dover, 1969.

- [301] S. Sahni y T. González. Problemas de aproximación P-completo. Diario de la ACM, 23(3):555–565, 1976.
- [302] A. Schönhage, M. Paterson y N. Pippenger. Encontrar la mediana. Revista de Ciencias de la Computación y Sistemas, 13(2):184–199, 1976.
- [303] Alexander Schrijver. Teoría de la Programación Lineal y Entera. John Wiley e hijos, 1986.
- [304] Alexander Schrijver. Caminos y flujos: un estudio histórico. CWI Trimestral, 6(3):169–183, 1993.
- [305] Roberto Sedgewick. Implementación de programas Quicksort. Comunicaciones de la ACM, 21(10):847–857, 1978.
- [306] Roberto Sedgewick. Algoritmos. Addison-Wesley, segunda edición, 1988.
- [307] Robert Sedgewick y Philippe Flajolet. Una introducción al análisis de algoritmos. Addison-Wesley, 1996.
- [308] Raimund Seidel. Sobre el problema del camino más corto de todos los pares en grafos no dirigidos no ponderados. Revista de Ciencias de la Computación y Sistemas, 51(3):400–403, 1995.
- [309] Raimund Seidel y CR Aragón. Árboles de búsqueda aleatorios. Algorítmica, 16(4–5):464–497, 1996.
- [310] Jo˜ao Setubal y Jo˜ao Meidanis. Introducción a la Biología Molecular Computacional. PWS Editorial, 1997.
- [311] Clifford A. Shaffer. Una introducción práctica a las estructuras de datos y el análisis de algoritmos. Prentice Hall, segunda edición, 2001.
- [312] Jeffrey Shallit. Orígenes del análisis del algoritmo euclidianos. historia matematica, 21(4):401–419, 1994.
- [313] Michael I. Shamos y Dan Hoey. Problemas de intersección geométrica. En Actas del 17º Simposio Anual sobre Fundamentos de Ciencias de la Computación, páginas 208–215, 1976.
- [314] M. Sharir. Un algoritmo de conectividad fuerte y sus aplicaciones en el análisis de flujo de datos. Com computadoras y matemáticas con aplicaciones, 7(1):67–72, 1981.
- [315] David B. Shmoys. Cálculo de soluciones casi óptimas para problemas de optimización combinatoria. En William Cook, L’aszl’o Lov’asz y Paul Seymour, editores, Combinatorial Optimization, volumen 20 de la Serie DIMACS en Matemáticas Discretas e Informática Teórica. Sociedad Matemática Estadounidense, 1995.
- [316] Avi Shoshan y Uri Zwick. Todos los pares de rutas más cortas en gráficos no dirigidos con pesos enteros. En Proceedings of the 40th Annual Symposium on Foundations of Computer Science, páginas 605–614, 1999.
- [317] Michael Sipser. Introducción a la Teoría de la Computación. tecnología del curso thomson, segunda edición, 2006.
- [318] Steven S. Skiena. Manual de diseño de algoritmos. Springer, segunda edición, 1998.
- [319] Daniel D. Sleator y Robert E. Tarjan. Una estructura de datos para árboles dinámicos. Diario de Ciencias informáticas y de sistemas, 26(3):362–391, 1983.

- [320] Daniel D. Sleator y Robert E. Tarjan. Árboles de búsqueda binarios autoajustables. Diario de la ACM, 32(3):652–686, 1985.
- [321] Joel Spencer. Diez conferencias sobre el método probabilístico, volumen 64 de la serie de conferencias regionales CBMS-NSF en matemáticas aplicadas. Sociedad de Matemáticas Industriales y Aplicadas, 1993.
- [322] Daniel A. Spielman y Shang-Hua Teng. Análisis suavizado de algoritmos: por qué el algoritmo simple generalmente toma tiempo polinomial. Revista de la ACM, 51(3):385–463, 2004.
- [323] Gilbert Strang. Introducción a las Matemáticas Aplicadas. Wellesley-Cambridge Press, 1986.
- [324] Gilbert Strang. Álgebra lineal y sus aplicaciones. Thomson Brooks/Cole, cuarta edición, 2006.
- [325] Volker Strassen. La eliminación gaussiana no es óptima. Numerische Mathematik, 14(3):354–356, 1969.
- [326] TG Szymanski. Un caso especial del problema de subsecuencia común máxima. Informe técnico TR-170, Laboratorio de Ciencias de la Computación, Universidad de Princeton, 1975.
- [327] Robert E. Tarjan. Primeros algoritmos de búsqueda en profundidad y gráficos lineales. Revista SIAM en Compuerta, 1(2):146–160, 1972.
- [328] Robert E. Tarjan. Eficiencia de un buen algoritmo de unión de conjuntos, pero no lineal. Diario de la ACM, 22(2):215–225, 1975.
- [329] Robert E. Tarjan. Una clase de algoritmos que requieren un tiempo no lineal para mantener conjuntos disjuntos. Revista de Ciencias de la Computación y Sistemas, 18(2):110–127, 1979.
- [330] Robert E. Tarjan. Estructuras de datos y algoritmos de red. Sociedad para la Industria y Matemáticas Aplicadas, 1983.
- [331] Robert E. Tarjan. Complejidad computacional amortizada. SIAM Journal on Algebraic and Métodos discretos, 6(2):306–318, 1985.
- [332] Robert E. Tarjan. Apuntes de clase: Unión de conjuntos disjuntos. COS 423, Universidad de Princeton, 1999.
- [333] Robert E. Tarjan y Jan van Leeuwen. Análisis del peor de los casos de algoritmos de unión de conjuntos. Journal of the ACM, 31(2):245–281, 1984.
- [334] George B. Thomas, Jr., Maurice D. Weir, Joel Hass y Frank R. Giordano. Tomás Cálculo. Addison-Wesley, undécima edición, 2005.
- [335] Mikkel Thorup. Clasificación determinista más rápida y colas de prioridad en el espacio lineal. En Actas del 9º Simposio ACM-SIAM sobre algoritmos discretos, páginas 550–555, 1998.
- [336] Mikkel Thorup. Rutas más cortas de fuente única no dirigidas con pesos enteros positivos en tiempo lineal. Revista de la ACM, 46(3):362–394, 1999.
- [337] Mikkel Thorup. En las colas de prioridad de RAM. SIAM Journal on Computing, 30(1):86–109, 2000.
- [338] Richard Tolimieri, Myoung An y Chao Lu. Matemáticas de Algoritmos de Transformada de Fourier Multidimensional. Springer, segunda edición, 1997.
- [339] P. van Emde Boas. Preservar el orden en un bosque en menos de un tiempo logarítmico. En Actas del 16º Simposio Anual sobre Fundamentos de Ciencias de la Computación, páginas 75–84, 1975.

- [340] P. van Emde Boas. Preservar el orden en un bosque en menos de un tiempo logarítmico y un espacio lineal. *Cartas de procesamiento de información*, 6(3):80–82, 1977.
- [341] P. van Emde Boas, R. Kaas y E. Zijlstra. Diseño e implementación de una cola de prioridad eficiente. *Teoría de sistemas matemáticos*, 10(1):99–127, 1976.
- [342] Jan van Leeuwen, editor. *Manual de Ciencias de la Computación Teórica, Volumen A: Algoritmos y Complejidad*. Elsevier Science Publishers y MIT Press, 1990.
- [343] Préstamo de Charles Van. Marcos computacionales para la transformada rápida de Fourier. Sociedad de Matemáticas Industriales y Aplicadas, 1992.
- [344] Robert J. Vanderbei. *Programación Lineal: Fundamentos y Extensiones*. Editores académicos de Kluwer, 1996.
- [345] Vijay V. Vazirani. *Algoritmos de aproximación*. Springer, 2001.
- [346] Rakesh M. Verma. Técnicas generales de análisis de algoritmos recursivos con aplicaciones. *SIAM Journal on Computing*, 26(2):568–581, 1997.
- [347] Hao Wang y Bill Lin. Árbol Pipelined van Emde Boas: algoritmos, análisis y aplicaciones. En 26th IEEE International Conference on Computer Communications, páginas 2471–2475, 2007.
- [348] Antonio F. Ware. Transformadas rápidas aproximadas de Fourier para datos espaciados irregularmente. *SIAM Revisión*, 40(4):838–856, 1998.
- [349] Stephen Warshall. Un teorema sobre matrices booleanas. *Revista de la ACM*, 9(1):11–12, 1962.
- [350] Michael S. Waterman. *Introducción a la Biología Computacional, Mapas, Secuencias y Genomas*. Chapman & Hall, 1995.
- [351] Marcos Allen Weiss. *Estructuras de datos y resolución de problemas usando C++*. Addison Wesley, segunda edición, 2000.
- [352] Marcos Allen Weiss. *Estructuras de datos y resolución de problemas usando Java*. Addison-Wesley, tercera edición, 2006.
- [353] Marcos Allen Weiss. *Estructuras de datos y análisis de algoritmos en C++*. Addison-Wesley, tercero edición, 2007.
- [354] Marcos Allen Weiss. *Estructuras de datos y análisis de algoritmos en Java*. Addison Wesley, segunda edición, 2007.
- [355] Hassler Whitney. Sobre las propiedades abstractas de la dependencia lineal. *Revista estadounidense de matemáticas*, 57 (3): 509–533, 1935.
- [356] Herbert S. Wilf. *Algoritmos y Complejidad*. AK Peters, segunda edición, 2002.
- [357] JWJ Williams. Algoritmo 232 (HEAPSORT). *Comunicaciones de la ACM*, 7(6):347–348, 1964.
- [358] Shmuel Winograd. Sobre la complejidad algebraica de las funciones. En *Actes du Congrès International des Mathématiciens*, volumen 3, páginas 283–288, 1970.
- [359] Andrés C.-C. Yao. Un límite inferior para encontrar círculos convexos. *Revista de la ACM*, 28(4):780–787, 1981.
- [360] Chee Yap. Una verdadera aproximación elemental a la recurrencia maestra y las generalizaciones. Manuscrito no publicado. Disponible en <http://cs.nyu.edu/yap/papers/>, julio de 2008.

1250

Bibliografía

- [361] Yinyu Ye. Algoritmos de Punto Interior: Teoría y Análisis. John Wiley & Sons, 1997.
- [362] Daniel Zwillinger, editor. Tablas y fórmulas matemáticas estándar de CRC. Chapman & Hall/CRC Press, 31<sup>a</sup> edición, 2003.

# Índice

Este índice utiliza las siguientes convenciones. Los números están ordenados alfabéticamente como si estuvieran escritos; por ejemplo, "árbol 2-3-4" se indexa como si fuera "árbol dos-tres-cuatro". Cuando una entrada se refiere a un lugar que no es el texto principal, el número de página va seguido de una etiqueta: ej. para el ejercicio, pr. para problema, fig. para figura, y n. para nota al pie. Un número de página etiquetado a menudo indica la primera página de un ejercicio o problema, que no es necesariamente la página en la que realmente aparece la referencia.

- $\cdot\text{n}/$ , 574
- $\gamma$  (proporción áurea), 59, 108 pr.
- $\gamma$  (conjuguado de la proporción áurea), 59 .n/
- (función phi de Euler), 943 .n/-algoritmo
- de aproximación, 1106, 1123 notación o, 50–51, 64
- notación O, 45 fig., 47–48, 64
- O0 -notación, 62 pr.
- $\overset{m}{}$  Notación O, 62 pr. !-
- notación, 51
  - notación, 45 fig., 48–49, 64
  - 1 -notación, 62 pr.
  - $\overset{m}{}$  -notación, 62 pr.
  - notación „, 44–47, 45 fig., 64
  - $\overset{m}{}$  -notación, 62 pr. fg
  - (conjunto), 1158 2
  - (miembro del conjunto), 1158
  - 62 (no es un miembro del conjunto), 1158 ;
  - (lenguaje vacío), 1058 (conjunto vacío), 1158
  - (subconjunto), 1159
  - (subconjunto propio), 1159
  - W (tal que), 1159 \
  - (establecer intersección), 1159
  - [ (establecer unión), 1159
- (diferencia establecida), 1159
- jj (valor de flujo), 710
- (longitud de una cadena), 986
- (cardinalidad establecida), 1161
- (producto cartesiano), 1162
- (producto cruzado), 1016 hi
- (secuencia), 1166
- (codificación estándar), 1057
- $\overset{\text{m}}{k}$  (elegir), 1185
- kk (norma euclidiana), 1222 Š
- (factorial), 57 de
- (techo), 54 bc (piso), 54 p # (raíz cuadrada inferior), 546 (raíz cuadrada superior), 546 p"
- P (suma), 1145 Q
- (producto), 1148 !
- (relación de adyacencia), 1169
- (relación de alcance), 1170 ^ (AND), 697, 1071 : (NOT), 1071 \_ (OR), 697, 1071 ^ (operador de grupo), 939 ^ (operador de convolución), 901

- (operador de cierre), 1058
- j (relación divide), 927 –
  - (relación no divide), 927
    - (módulo n equivalente), 54, 1165 ej.
    - 6 (módulo n no equivalente), 54 CEn
    - (clase de equivalencia módulo n), 928 Cn
    - (módulo n de suma), 940 n (módulo n de multiplicación), 940 . " (cadena  $\frac{a}{pq}$  / (símbolo Legendre)), 982 pr.
    - vacía), 986, 1058 (relación de prefijo), 986 (relación de sufijo), 986  $<_x$  (relación anterior), 1022 // (símbolo de comentario), 21 (relación mucho mayor que), 574 (relación mucho menor que), 783 P (relación de reducibilidad polinomial-tiempo), 1067, 1077 ej.
- AA-tree, 338
- grupo abeliano, 940
- ARRIBA, 1024
- relación anterior ( $<_x$ ), 1022 hijo ausente, 1178 series
- absolutamente convergentes, 1146 leyes de absorción para conjuntos, 1160 problema abstracto, 1054
- par aceptable de enteros, 972
- aceptación
  - por un algoritmo, 1058 por un autómata finito, 996 estado de aceptación, 995
- método de contabilidad, 456–459
  - para contadores binarios, 458 para tablas dinámicas, 465–466 para operaciones de pila, 457–458, 458 ej.
- Función de Ackermann, 585
- problema de selección de actividad, 415–422, 450 gráfico acíclico, 1170 relación con matroides, 448 pr. añadir instrucción, 23 suma
  - de enteros binarios, 22 ej. de matrices, 1220
  - módulo n (Cn), 940 de polinomios, 898 grupo
  - aditivo módulo n, 940
- direcciónamiento, abierto, ver tabla hash de direcciones abiertas
- AGREGAR SUBARRAY, 805
- pr. representación de lista de adyacencia, 590 reemplazada por una tabla hash, 593 ex. representación de matriz de adyacencia, 591 relación de adyacencia (!), 1169 vértices adyacentes, 1169 borde admisible, 749 red admisible, 749–
- 750 adversario, 190 análisis agregado, 452–456
  - para contadores binarios, 454–455 para búsqueda primero en amplitud, 597 para búsqueda primero en profundidad, 606 para el algoritmo de Dijkstra, 661 para estructuras de datos de conjuntos disjuntos, 566–567, 568 ej.
  - para tablas dinámicas, 465
  - para montones de Fibonacci, 518, 522
  - ex. para el escaneo de Graham, 1036 para el algoritmo de Knuth-Morris-Pratt, 1006 para el algoritmo de Prim, 636 para corte de barras, 367 para las rutas más cortas en un dag, 655 para operaciones de pila, 452–454 flujo agregado, 863 método Akra-Bazzi para resolver una recurrencia, 112–113
- algoritmo, 5
  - corrección de, 6
  - origen de la palabra, 42 tiempo de ejecución de, 25 como tecnología, 13 Alice, 959 ALLOCATE-NODE, 492
- ALLOCATE-OBJECT, 244
- asignación de objetos, 243–244
- caminos más cortos de todos los pares, 644, 684–707 en gráficos dinámicos, 707 en gráficos densos, 706 pr. Algoritmo de Floyd-Warshall para, 693–697, 706 Algoritmo de Johnson para, 700–706 por multiplicación de matrices, 686–693, 706–707 por elevación repetida al cuadrado, 689–691 alfabeto, 995, 1057 .n/, 574 análisis amortizado, 451–478
  - método contable de, 456–459 análisis agregado, 367, 452–456

- para permutación de inversión de bits, 472 pr.
  - para búsqueda primero en amplitud, 597 para búsqueda primero en profundidad, 606 para el algoritmo de Dijkstra, 661 para estructuras de datos de conjuntos disjuntos, 566–567, 568 ej., 572 ej., 575–581, 581–582 ej. para tablas dinámicas, 463–471 para montones de Fibonacci, 509–512, 517–518, 520–522, 522 ej. para el algoritmo genérico push-relabel, 746 para el escaneo de Graham, 1036 para el algoritmo Knuth-Morris-Pratt, 1006 para hacer dinámica la búsqueda binaria, 473 pr. método potencial de, 459–463 para la reestructuración de árboles rojo-negros, 474 pr. para listas autoorganizadas con mover al frente, 476 pr.
  - para rutas más cortas en un dag, 655
  - para pilas en almacenamiento secundario, 502 pr.
  - para árboles de peso equilibrado, 473 pr.
  - coste amortizado
    - en el método contable, 456 en análisis agregado, 452 en el método potencial, 459 antepasado, 1176
  - menos común, 584
    - pr.
  - Función Y (^), 697, 1071 Puerta Y, 1070 y, en
  - pseudocódigo, 22 aristas
  - antiparalelas, 711–712 relación antisimétrica, 1164 ANY SEGMENTS-INTERSEC, 1025 aproximación por mínimos cuadrados, 835–839 de suma por integrales, 1154 –1156 algoritmo de aproximación, 10, 1105–1140
  - para embalaje en contenedores, 1134 pr. para satisfacibilidad MAX-CNF, 1127 ej. para camarilla máxima, 1111 ex., 1134 pr.
  - para coincidencia máxima, 1135 pr. para árbol de expansión máxima, 1137 pr. para corte de peso máximo, 1127 ex. para la satisfacibilidad de MAX-3-CNF, 1123–1124,
- 1139
- para cobertura de vértice de peso mínimo, 1124–1127, 1139 para la programación de máquinas en paralelo, 1136 pr. aleatorio, 1123
  - para cobertura de conjunto, 1117–1122, 1139 para suma de subconjunto, 1128–1134, 1139 para problema del viajante de comercio, 1111–1117, 1139
  - para cubierta de vértices, 1108–1111, 1139
  - para cubierta de conjunto ponderado, 1135 pr. para 0-1 problema de mochila, 1137 pr., 1139
  - error de aproximación, 836
  - relación de aproximación, 1106, 1123
  - esquema de aproximación, 1107
  - APROX-MIN-WEIGHT-VC, 1126 APROX-SUBSET-SUM, 1131 APROX-TSP-TOUR, 1112 APROX -VERTEX-COVER, 1109 arbitraje, 679 pr. arco, ver argumento de borde de una función, 1166–1167 instrucciones aritméticas, 23 aritmética modular, 54, 939–946 serie aritmética, 1146 aritmética con infinitos, 650 brazo, 485 matriz, 21 Monge, 110 pr. pasando como parámetro, 21
  - punto de articulación, 621 pr.
  - asignación
    - múltiple, 21
    - satisfactoria, 1072, 1079
    - verdad, 1072, 1079
  - leyes asociativas para conjuntos, 1160 operación asociativa, 939
  - asintóticamente mayor, 52 asintóticamente no negativa, 45 asintóticamente positiva, 45 asintóticamente menor, 52 límite asintóticamente estrecho, 45 eficiencia asintótica, 43 asintótica límite inferior, 48 notación asintótica, 43–53, 62 pr. y algoritmos gráficos, 588 y
    - linealidad de sumas, 1146 límite superior asintótico, 47 atributo de un objeto, 21 aumento de un flujo, 716
  - estructuras de datos que aumentan, 339–355 ruta de aumento, 719–720, 763 pr. autenticación, 284 pr., 960–961, 964

- autómata  
 finito, 995  
 coincidencia de cadenas, 996–  
 1002 función hash auxiliar, 272  
 programa lineal auxiliar, 886 tiempo  
 de ejecución de caso promedio, 28, 116 AVL-  
 INSERT, 333 pr.  
 Árbol AVL, 333 pr., 337  
 axiomas, para probabilidad, 1190
- cara de bebé, 602 ej.  
 borde posterior, 609, 613  
 sustitución posterior, 817  
**INSTANCIA DE CUBIERTA DE MAL AJUSTE**, 1122 ex.
- SALDO**, 333 pr. árbol  
 de búsqueda equilibrado  
     Árboles AA, 338  
     árboles AVL, 333 pr., 337  
     árboles B, 484–504  
     árboles vecinos k, 338  
     árboles rojo-negro, 308–338  
     árboles de chivo expiatorio,  
         338 árboles splay, 338,  
         482 treaps, 333 pr., 338  
     2-3-4 árboles, 489, 503 pr. 2-3  
     árboles, 337, 504  
     árboles de peso balanceado, 338, 473 pr.  
 bolas y contenedores, 133–134, 1215 pr.  
 base-a pseudoprimo, 967 caso  
 base, 65, 84 base, en  
 ADN, 391 solución  
 factible básica, 866 solución básica,  
 866 variable básica, 855  
 función base, 835  
 Teorema de Bayes, 1194  
**BELLMAN-FORD**, 651  
 Algoritmo de Bellman-Ford ,  
 651 –655, 682 para caminos más cortos de todos  
 los pares, 684 en el algoritmo de  
 Johnson, 702–704 y funciones objetivas,  
 670 ej. para resolver sistemas de  
 restricciones en diferencias,  
 668  
 Mejora del yen a 678 pr.  
**ABAJO**, 1024  
 Prueba de Bernoulli, 1201  
 y bolas y cubos, 133–134 y rayas,  
 135–139
- tiempo de ejecución en el mejor de los  
 casos, 29  
 ej., 49 BFS, 595 BIASED-RANDOM,  
 117 ej. componente biconectado, 621 pr.  
 notación big-oh, 45 fig., 47–48, 64 notación  
 big-omega, 45 fig., 48–49, 64 función biyectiva,  
 1167 código de carácter  
 binario, 428 contador binario  
 analizado por  
     método contable, 458 analizado por análisis  
     agregado, 454–455 analizados por el método  
     potencial, 461–462 bits invertidos, 472 pr. función  
     de entropía binaria, 1187  
 algoritmo gcd binario, 981 pr. montón  
 binario, ver relación binaria montón,  
 1163 búsqueda binaria, 39  
 ej. con inserción rápida,  
 473 pr. en ordenación por  
     inserción, 39 ej. en la fusión de  
     subprocesos múltiples, 799–  
     800 en la búsqueda de árboles B, 499 ej.
- BÚSQUEDA BINARIA**, 799  
 árbol de búsqueda binaria, 286–307  
     árboles AA, 338  
     árboles AVL, 333 pr., 337  
     eliminación de, 295–298, 299 ex. con  
     llaves iguales, 303 pr. inserción  
         en, 294–295 k-árboles vecinos,  
         338 clave máxima de, 291  
         clave mínima de, 291 óptima,  
         397–404, 413 predecesor  
         en, 291–292 consultas, 289–  
         294 construido aleatoriamente,  
         299–303, 304 pr.  
     conversión a la derecha de, 314 ej. árboles  
     de chivos expiatorios, 338  
     búsqueda, 289–291 para  
     clasificación, 299 ex.  
     árboles esparcidos, 338  
     sucesor en, 291–  
     292 y treaps, 333 pr. árboles  
     de peso equilibrado,  
         338 véase también la propiedad  
         del árbol de búsqueda binaria  
     del árbol rojo-negro, 287 en treaps, 333  
     pr. vs propiedad min-  
     heap, 289 ex.

- árbol binario, 1177
  - completo,
  - 1178 número de diferentes, 306 pr.
  - representación de, 246
  - superpuesta a un vector de bits, 533–534 véase también coeficiente binomial del
- árbol de búsqueda binaria, 1186–1187
- distribución binomial, 1203–1206 y bolas y
  - contenedores, 133 valor
  - máximo de, 1207 ej. colas de, 1208–1215 expansión binomial,
- 1186 montón binomial, 527 pr.
- árbol binomial, 527 pr.
- embalaje bin., 1134 pr.
- gráfico bipartito, 1172 red
- de flujo correspondiente
  - de, 732 d-regular, 736 ex. e hipergrafías, 1173
  - ej. coincidencia
  - bipartita, 530, 732–736, 747 ej.,
- 766 Algoritmo de Hopcroft-Karp para, 763 pr.
- paradoja del cumpleaños, 130–133, 142 ej.
- bisección de un árbol, 1181 pr.
- problema bitónico euclíadiano del viajante de comercio, 405 pr.
- secuencia bitónica, 682 pr.
- recorrido bitónico, 405
- pr. operación de bits,
  - 927 en el algoritmo de Euclides, 981
  - pr. permutación de inversión de bits, 472 pr., 918 BIT-REVERSE-COPY, 918
  - contador binario invertido de bits, 472 pr.
  - BIT-REVERSED-INCREMENT, 472 pr. vector de bits, 255 ej., 532–536 altura negra, 309 vértice negro, 594, 603 flujo de bloqueo, 765 estructura de bloque en pseudocódigo, 20 Bob, 959 desigualdad de Boole, 1195 ej. circuito combinacional booleano, 1071 elemento combinacional booleano, 1070 conectivo booleano, 1079 fórmula booleana, 1049, 1066 ej., 1079, 1086 ej.
  - función booleana, 1187 ej. multiplicación de matrices booleanas, 832 ej.
- Algoritmo de Borúuvka, 641
- árbol de expansión de cuello de botella, 640 pr. problema del viajante de comercio con cuello de botella, 1117 ej. fondo de una pila, 233 BOTTOM-UP-CUT-ROD, 366 método de abajo hacia arriba, para programación dinámica, 365
- atado
  - asintóticamente ajustado, 45
  - asintótica inferior, 48
  - asintótica superior, 47 en coeficientes binomiales, 1186–1187 en distribuciones binomiales, 1206 polilogarítmica, 57 en las colas de una distribución binomial, 1208–1215 ver también límites inferiores
- condición límite, en una recurrencia, 67, 84 límite de un polígono, 1020 ej. delimitando una sumatoria, 1149–1156 cuadro, anidamiento, 678 pr.
- Árbol BC, 488
- factor de ramificación, en árboles B, 487
- instrucciones de bifurcación, 23 búsqueda en amplitud, 594–602, 623 en flujo máximo, 727–730, 766 y caminos más cortos, 597–600, 644 similitud con el algoritmo de Dijkstra , 662, 663 ej.
- árbol primero en anchura, 594, 600 puente, 621 pr.
- Árbol B, 489 n.
- Árbol B, 484–504
  - comparado con árboles rojo-negro, 484, 490
  - creación, 492
  - eliminación de, 499–502 nodo completo en, 489 altura de, 489–490 inserción en, 493–497 grado mínimo de, 489 clave mínima de, 497 ej. propiedades de, 488–491 búsqueda, 491–492 dividir un nodo en, 493–495 2-3-4 árboles, 489 B-TREE-CREATE, 492 B-TREE-DELETE, 499 B-TREE-INSERT, 495

- B-ÁRBOL-INSERCIÓN-NO COMPLETO,  
496 B-ÁRBOL-BÚSQUEDA, 492, 499 ej.  
B-TREE-SPLIT-CHILD, 494  
BUBBLESORT, 40 pr.  
cubeta, 200  
clasificación de cubeta, 200–  
204 CLASIFICACIÓN DE  
CUBETA, 201 BUILD-MAX-HEAP,  
157 BUILD-MAX-HEAP0 , 167 pr.  
BUILD-MIN-HEAP, 159 operación  
mariposa, 915 por, en  
pseudocódigo, 21  
  
caché, 24, 449 pr. golpe  
de caché, 449 pr.  
señorita caché, 449 pr.  
olvido de caché, 504  
almacenamiento en caché, fuera  
de línea, 449 pr. llamar  
en un cálculo de subprocessos múltiples, 776 de  
una subrutina, 23, 25 n. por valor,  
21 borde de  
llamada, 778 lema  
de cancelación, 907 cancelación  
de flujo, 717 forma canónica para  
programación de tareas, 444 capacidad de un corte,  
721 de un  
borde, 709  
residual, 716, 719  
de un vértice, 714 ex.  
restricción de capacidad,  
709–710 cardinalidad de un conjunto  
(jj), 1161 número de Carmichael, 968,  
975 ex.  
Producto cartesiano (), 1162 Suma  
cartesiana, 906 ex. corte en  
cascada, 520 CORTE  
EN CASCADA, 519 números  
catalanes, 306 pr., 372 función de techo  
(de), 54 en teorema maestro, 103–  
106 instrucción de techo, 23 evento  
cierto, 1190 certificado  
  
en un criptosistema, 964 para  
algoritmos de verificación, 1063  
ELIMINACIÓN DE HASH ENCADENADO, 258  
INSERCIÓN DE HASH ENCADENADO, 258  
  
CHAINED-HASH-SEARCH, 258  
encadenamiento, 257–260, 283 pr.  
cadena de un casco convexo, 1038  
cambiando una llave, en un montón de Fibonacci, 529 pr.  
cambio de variables, en el método de sustitución, 86–87  
  
código de carácter, 428  
programa de juego de ajedrez, 790–791 niño  
  
en un árbol binario, 1178 en  
un cálculo de subprocessos múltiples, 776 en un  
árbol enraizado, 1176 lista  
de niños en un montón de Fibonacci, 507  
teorema del resto chino, 950–954, 983 chip multiprocesador,  
772 transformación chirp, 914 ex.  
n elige 1185 k , acorde, 345 ej.  
  
Cilk, 774, 812 Cilk+  
+, 774, 812 texto  
cifrado, circuito 960  
  
combinacional booleano, 1071  
profundidad de,  
919 para transformada rápida de Fourier, 919–920  
CIRCUIT-SAT, 1072  
satisfacibilidad del circuito, 1070–1077  
circular, lista doblemente enlazada con un centinela, 239 lista  
enlazada circular, 236 véase  
también lista enlazada  
clase  
complejidad, 1059  
equivalencia, 1164  
clasificación de aristas en  
búsqueda en anchura, 621 pr. búsqueda  
en profundidad primero, 609–610, 611 ej. en un dag  
de subprocessos múltiples, cláusula 778–  
779, área limpia 1081–  
1082, 208 pr. camarilla,  
1086–1089, 1105 algoritmo de  
aproximación para, 1111 ex., 1134 pr.  
  
CLIQUE, 1087  
intervalo cerrado, 348  
semicírculo cerrado, 707 par  
más cercano, búsqueda, 1039–1044, 1047 heurística  
del punto más cercano, 1117 ej.

- propiedad
  - de grupo de cierre, 939
  - de un idioma, 1058
  - operador (), 1058
  - transitivo, ver clúster de cierre
- transitivo
  - en un vector de bits con un árbol superpuesto de altura constante, 534
  - para computación paralela, 772
  - en estructuras proto van Emde Boas, 538 en árboles van Emde Boas, 546
- agrupamiento,
  - 272 CNF (forma normal conjuntiva), 1049, 1082 CNF
  - satisfacibilidad, 1127 ex.
  - engrosamiento de hojas de recursividad en ordenación por fusión, 39 pr. cuando se genera recursivamente, código 787,
  - 428–429 Huffman,
  - 428–437, palabra de código
    - 450, codominio
    - 429, binomial de coeficiente
    - 1166, 1186 de un polinomio, 55, 898 en forma floja, representación de coeficiente 856–890 820, 834 y multiplicación rápida, cofactor 903–905, cambio de moneda 1224, 446 pr. colinealidad, 1016 colisión, 257 resolución por encadenamiento, 257–260 resolución por direccionamiento abierto, 269–277 función hash resistente a colisiones, 964 colores, 1103 pr., 1180 pr. color, de un nodo rojo-negro-árbol, 308 columna-orden mayor, 208 pr. rango de columna, 1223 clasificación de columna, 208 pr. vector de columna, 1218 combinación, 1185 circuito combinacional, 1071 elemento combinacional, 1070 paso combinado, en divide y vencerás, 30, 65 comentario, en pseudocódigo (//), 21 mercancía , 862 divisor común, 929 mayor, ver mayor común divisor
  - múltiplo común, 939 ej.
  - subexpresión común, 915
  - subsecuencia común, 7, 391 más larga, 7, 390–397, 413 leyes comutativas para conjuntos, 1159 operación comutativa, 940 COMPACTIFY-LIST, 245 ex. lista compacta, 250 pr. BÚSQUEDA EN LISTA COMPACTA, 250 pr. COMPACT-LIST-SEARCH0, 251 años segmentos de línea comparables, 1022 COMPARE-EXCHANGE, 208 pr. operación de intercambio de comparación, 208 pr. clasificación por comparación, 191 y árboles de búsqueda binarios, 289 ej. aleatorio, 205 pr. y selección, 222 actividades compatibles, 415 matrices compatibles, 371, 1221 análisis competitivo, 476 pr. complemento de un evento, 1190 de un gráfico, 1090 de una lengua, 1058 de un conjunto, 1160 holgura complementaria, 894 pr. gráfico completo, 1172 árbol k-ario completo, 1179 véase también montón integridad de un idioma, 1077 ej. paso completo, 782 tiempo de finalización, 447 pr., 1136 pr. clase de complejidad, 1059 co-NP, 1064 NP, 1049, 1064 NPC, 1050, 1069 P, 1049, 1055 medida de complejidad, 1059 matrices de inversión de números complejos de, 832 ej. multiplicación de, 83 ej. raíz compleja de la unidad, 906 interpolación en, 912–913 componente biconectado, 621 pr. conectado, 1170 fuertemente con-

- gráfico de componentes, 617 número compuesto, 928 testigo, 968 composición, de cálculos multiproceso, 784 fig.
- profundidad computacional, 812 geometría computacional, 1014–1047 problema computacional, 5–6 dag de computación, 777 computación, multiproceso, 777 COMPUTE-PREFIX-FUNCTION, 1006 COMPUTE-TRANSITION-FUNCTION, 1001 concatenación
  - de idiomas, 1058 de cadenas, 986
- problema concreto, 1055 palabras clave de concurrencia, 774, 776, 785 plataforma de concurrencia, 773 instrucción de rama condicional, 23 independencia condicional, 1195 ex. probabilidad condicional, 1192, 1194 configuración, 1074 conjugada de la proporción áurea (y), 59 transpuesta conjugada, 832 ej. forma normal conjuntiva, 1049, 1082 componente conectado, 1170 identificado mediante búsqueda en profundidad, 612 ej. identificado usando estructuras de datos de conjuntos disjuntos, 562–564
- COMPONENTES CONECTADOS, 563 gráfico conectado, 1170 conectivo, 1079 co- NP (clase de complejidad), 1064 paso de conquista, en divide y vencerás, 30, 65 conservación del flujo, 709–710 consistencia de literales, 1088 secuencial, 779, 812 CONSOLIDATE, 516 consolidación de una lista raíz de montón de Fibonacci, 513–517 restricción, 851 diferencia, 665 igualdad, 670 ej., 852–853 desigualdad, 852–853 lineal, 846 no negatividad, 851, 853 ajustada, 865 violación de, 865 gráfico de restricción, 666–668 contiene, en un camino, 1170 borde de continuación, 778 distribución de probabilidad uniforme continua, 1192 contracción
  - de una tabla dinámica, 467–471 de una matríoide, 442 de un gráfico no dirigido por una arista, 1172
- instrucciones de control, 23 propiedad de convergencia, 650, 672–673 serie convergente, 1146 conversión de binario a decimal, 933 ej. combinación convexa de puntos, 1015 función convexa, 1199 casco convexo, 8, 1029–1039, 1046 pr. capas convexas, 1044 pr. polígono convexo, 1020 ej. conjunto convexo, 714 ej. convolución ('), 901 teorema de convolución, 913 instrucción de copia, 23 corrección de un algoritmo, 6 red de flujo correspondiente para emparejamiento bipartito, 732 conjunto numerable infinito, 1161 contador, ver conteo de contador binario, 1183–1189 probabilístico, 143 pr. ordenación por conteo, 194–197 en ordenación radix, 198 CLASIFICACIÓN POR CONTAJE, 195 problema del coleccionista de cupones, 134 cubrir camino, 761 pr. por un subconjunto, 1118 vértice, 1089, 1108, 1124–1127, 1139 encubierto, 1024 CREAR-NUEVO-RS-VEB-ÁRBOL, 557 pr. crédito, 456 borde crítico, 729 ruta crítica de un dag, 657 de un cálculo de subprocessos múltiples, 779 cruzar un corte, 626 borde cruzado, 609 producto cruzado (), 1016

- cryptosystem, 958–965, 983 spline
- cúbico, 840 pr. cambio
- de moneda, 390 ex., 679 pr. ajuste de
- curvas, 835–839
- cortar
  - capacidad de, 721
  - en cascada, 520
  - de una red de flujo, 720–724
  - mínimo, 721, 731 ex. flujo
  - neto a través, 720 de un
  - gráfico no dirigido, 626 peso de, 1127 ex.
- CUT, 519
- CUT-ROD, 363
- corte, en un montón de Fibonacci, 519
- ciclo de un gráfico, 1170
  - hamiltoniano, 1049, 1061
  - peso medio mínimo, 680 pr. peso
  - negativo, ver ciclo de peso negativo y caminos más cortos, 646–647 grupo cíclico, 955
- rotación cíclica, 1012
- ex. ciclismo, de algoritmo
- simplex, 875
- dag, véase el gráfico acíclico dirigido
- DAG-SHORTEST-PATHS, 655 d-ary
- heap, 167 pr. en
  - algoritmos de caminos más cortos, 706 pr.
- instrucciones de movimiento de datos, 23 modelo de datos paralelos, 811 estructura de datos, 9, 229–355, 481–585 árboles AA, 338 aumento de, 339–355
- árboles AVL, 333 pr., 337
- árboles de búsqueda binaria, 286–307 montones binomiales, 527 pr. vectores de bits, 255 ej., 532–536 árboles B, 484–504 deque, 236 ej. diccionarios, 229 tablas de direcciones directas, 254–255 para conjuntos disjuntos, 561–585 para gráficos dinámicos, 483 conjuntos dinámicos, 229–231 árboles dinámicos, 482 árboles de búsqueda exponencial, 212, 483 Montones de Fibonacci, 505–530 árboles de fusión, 212 , 483/vértice, 1169
- tablas hash, 256–261
- montones, 151–169 árboles de intervalo, 348–354 árboles k-vecinos, 338 listas enlazadas, 236–241 montón fusionable, 505 árboles de estadísticas de orden, 339–345 persistentes, 331 pr., 482 potencial de , 459 colas prioritarias, 162–166 estructuras proto van Emde Boas, 538–545 colas, 232, 234–235 árboles radix, 304 pr. árboles rojo-negros, 308–338 montones relajados, 530 árboles enraizados, 246–249 árboles de chivo expiatorio, 338 en almacenamiento secundario, 484–487 listas de omisión, 338 árboles separados, 338, 482 pilas, 232–233 trampas, 333 pr., 338 2-3-4 montones, 529 pr. 2-3-4 árboles, 489, 503 pr. 2-3 árboles, 337, 504 árboles van Emde Boas, 531–560 árboles de peso equilibrado, 338 tipo de datos, 23 fecha límite, 444 desasignación de objetos, 243–244 decisión mediante un algoritmo, 1058–1059 problema de decisión, 1051, 1054 y problemas de optimización, 1051 árbol de decisión, 192–193 DECREASE-KEY, 162, 505 disminución de una clave en montones de Fibonacci, 519–522 en montones 2-3-4, 529 pr. DECRETO, 456 ej. degeneración, 874 grado
  - de una raíz de árbol binomial, 527 pr.
  - máximo, de un montón de Fibonacci, 509, 523–526
  - mínimo, de un árbol B, 489 de
  - un nodo, 1177 de
  - un polinomio, 55, 898 de un

- ligado a grados, 898  
**ELIMINAR**, 230, 505  
**ELIMINAR LA MITAD MÁS GRANDE**, 463  
 ex. supresión  
   de árboles de búsqueda binarios, 295–298, 299 ex. de  
   un vector de bits con un árbol binario superpuesto, 534 de  
     un vector  
   de bits con un árbol superpuesto de altura constante, 535 de  
     árboles B, 499–502 de  
   tablas hash encadenadas, 258  
   de tablas de direcciones directas, 254  
   de tablas dinámicas, 467 –471 de  
   montones de Fibonacci, 522, 526 pr. de  
   montones, 166 ej. de árboles de intervalo, 349  
   de listas enlazadas, 238  
   de tablas hash de direcciones  
     abiertas, 271 de árboles de  
     estadísticas de orden, 343–344 de estructuras  
   proto van Emde Boas, 544 de colas, 234 de  
   árboles rojo-negro, 323–330 de pilas , 232 de estados de  
   línea de barrido, 1024  
   de 2-3-4 montones, 529 pr. de los árboles  
   de van Emde Boas,  
     554–556
- Leyes de DeMorgan**  
   para lógica proposicional, 1083 para  
   conjuntos, 1160, 1162 ej.  
**gráfico denso**, 589  
   -denso, 706 pr.  
**densidad**  
   de números primos, 965–966 de una  
   barra, 370 ej.  
**variables**  
   aleatorias indicadoras y de dependencia, 119  
   lineales, 1223  
   ver también promedio de
- profundidad de independencia, de un nodo en un árbol de  
   búsqueda binario  
   construido  
   aleatoriamente, 304 pr. de un circuito, 919  
   de un nodo en un árbol enraizado, 1177 de  
   árbol de recursión de  
   clasificación rápida, 178 ej. de una pila, 188 pr.  
   problema de determinación  
   de la profundidad, 583 pr. bosque primero en profundidad, 603 búsqueda en profundidad, 603–612, 623
- en la búsqueda de puntos de articulación, puentes y  
   componentes biconectados, 621 pr. en la  
   búsqueda de componentes fuertemente conectados, 615–  
     621, 623 en  
   clasificación topológica, 612–615 árbol de  
   profundidad primero, 603  
   deque, 236 ej.  
**DEQUEUE**, 235  
   derivada de una serie, 1147  
   descendiente, 1176  
   vértice de destino, 644 det,  
   véase determinante  
   determinación carrera, 788  
   determinante, 1224–1225  
     y multiplicación de matrices, 832 ej. algoritmo  
   determinista, 123 multihilo, 787  
**BÚSQUEDA**  
**DETERMINISTA**, 143 pr.  
**DFS**, 604  
**DFS-VISIT**, 604 DFT  
   (transformada discreta de Fourier), 9, 909 matriz  
   diagonal, 1218 LUP  
   descomposición de, 827 ex. diámetro de  
   un árbol, 602 ex. diccionario, 229  
   restricciones de  
   diferencia, 664–670 ecuación de diferencia,  
   ver diferencia de recurrencia de conjuntos (),  
   1159 simétrica, 763 pr. diferenciación  
     de una serie, 1147 firma  
   digital, 960 dígrafo, véase gráfico dirigido  
**DIJKSTRA**, 658 Algoritmo  
   de Dijkstra, 658–664, 682  
   para rutas más cortas de todos los pares, 684,  
   704 implementado con un montón de Fibonacci, 662  
   implementado con un montón mínimo, 662 con  
   pesos de borde enteros, 664 ex. en el  
   algoritmo de Johnson, 702 similitud  
   con la búsqueda en amplitud, 662, 663 ej.
- similitud con el algoritmo de Prim, 634, 662  
**DIRECCIÓN DIRECTA-ELIMINAR**, 254  
 direcciónamiento directo, 254–255, 532–536  
**INSERCIÓN DE DIRECCIÓN DIRECTA**, 254  
**DIRECT-ADDRESS-SEARCH**, 254 tabla de  
 direcciones directas, 254–255 gráfico

- y bordes posteriores,  
613 y gráficos de componentes,  
617 y caminos hamiltonianos, 1066 ej.  
camino simple más largo en, 404 pr.  
para representar un cálculo de  
subprocesos múltiples,  
777 algoritmo de rutas más cortas de fuente única  
para, 655–658  
tipo topológico de, 612–615, 623 gráfico  
dirigido, 1168 caminos  
más cortos de todos los pares en, 684–707  
gráfico de restricción, 666  
recorrido de Euler, 623 pr., 1048  
ciclo hamiltoniano de, 1049 y  
caminos más largos, 1048  
cubierta de camino de, 761 pr.  
Diagrama PERT, 657, 657 ej.  
semitransitivo, 621 ej. ruta  
más corta de entrada,  
643 de una sola fuente rutas más cortas de entrada,  
643–683 conectadas  
individualmente, 612  
ex. cuadrado de, 593 ex.  
clausura transitiva de,  
697 trasposición de, 592 ex.  
Fregadero universal, 593 ej. véase también gráfico acíclico dirigido, red  
segmento dirigido, 1015–1017  
versión dirigida de un gráfico no dirigido, 1172  
DIRECCIÓN, 1018  
área sucia, 208 pr.  
DESCARGA, 751  
descarga de un vértice desbordante, 751 vértice  
descubierto, 594, 603 tiempo de  
descubrimiento, búsqueda en profundidad primero,  
605 transformada discreta de Fourier, 9,  
909 logaritmo discreto, 955  
teorema del logaritmo discreto, 955  
distribución de probabilidad discreta, 1191  
variable aleatoria discreta , 1196–1201  
estructura de datos de conjunto disjunto,  
561–585 análisis de, 575–581, 581  
ex. en componentes conectados, 562–564  
en determinación de profundidad, 583  
pr. implementación disjoint-set-forest de, 568–  
572 en el  
algoritmo de Kruskal, 631 caso  
especial de tiempo lineal de, 585  
implementación de lista enlazada de, 564–568  
en ancestros menos comunes fuera de línea, 584  
pr. en mínimo fuera de línea, 582  
pr. en programación de tareas,  
448 pr. bosque inconexo, 568–572  
análisis de, 575–581, 581 ej.  
propiedades de rango de, 575, 581 ex.  
ver también estructura de datos de conjunto disjunto  
conjuntos disjuntos,  
1161 forma normal disyuntiva, 1083  
disco, 1028 ex.  
unidad de disco, 485–  
487 véase también  
almacenamiento  
secundario DISK-  
READ, 487 DISK-WRITE, 487 distancia  
editar, 406 pr.  
euclidiana, 1039  
Lm, 1044 ej.  
Manhattan, 225 pr., 1044 ex. de  
un camino más corto, 597  
memoria distribuida, 772  
distribución  
binomial, 1203–1206  
continuo uniforme, 1192 discreto,  
1191 geométrico,  
1202–1203 de entradas,  
116, 122 de números  
primos, 965 probabilidad,  
1190 casco disperso,  
1046 pr. uniforme, 1191  
leyes distributivas para conjuntos,  
1160 series divergentes,  
1146 método divide y vencerás, 30–35, 65  
análisis de, 34–35  
para búsqueda binaria, 39 ej.  
para conversión de binario a decimal, 933 ex. para la  
transformada rápida de Fourier, 909–912  
para encontrar el par de puntos más cercano,  
1040–1043  
para encontrar la envolvente convexa,  
1030 para inversión de matriz, 829–  
831 para multiplicación de matriz, 76–83, 792–797  
para problema de subarreglo máximo, 68–75  
para clasificación por fusión, 30–37,  
797–805 para multiplicación, 920 pr .

- para la multiplicación de matrices multiproceso, 792–797
- para la clasificación por fusión multiproceso, 797–805 para la clasificación rápida, 170–190 relación con la programación dinámica, 359 para la selección, 215–224 resolución de recurrencias para, 83–106, 112–113 para el algoritmo de Strassen, 79–83 instrucción de división, 23 relación de división ( $\mid$ ), 927 paso de división, en divide y vencerás, 30, 65 método de división, 263, 268–269 ej. teorema de la división, 928 divisor, 927–928 común, 929 véase también máximo común divisor DNA, 6–7, 390–391, 406 pr. DNF (forma normal disyuntiva), 1083 relación no divide ( $\nmid$ ), 927 dominio, 1166 relación domina, 1045 pr. hash doble, 272–274, 277 ex. lista doblemente enlazada, 236 véase también lista enlazada hasta, en pseudocódigo, 21 grafo regular d, 736 ej. dualidad, 879–886, 895 pr. débil, 880–881, 886 ej. programa lineal dual, 879 tecla ficticia, 397 gráfico dinámico, 562 n.
- algoritmos de rutas más cortas de todos los pares para, 707 estructuras de datos para, 483 algoritmo de árbol de expansión mínimo para, 637 ej. cierre transitivo de, 705 pr., 707 algoritmo dinámico de subprocessos múltiples, consulte algoritmo multiproceso multiproceso dinámico, 773 estadísticas de orden dinámico, 339–345 método de programación dinámica, 359–413 para selección de actividad, 421 ej. para los caminos más cortos de todos los pares, 686–697 para el problema del viajante euclíadiano binómico, 405 pr. de abajo hacia arriba, 365 por romper una cuerda cónicoparabólica, 778 capacidad de, 709 clasificación en comparación con algoritmos codiciosos, 381, 390 ej., 418, 423–427 para la distancia de edición, 406 pr. elementos de, 378–390 para el algoritmo de Floyd-Warshall, 693–697 para la planificación del inventario, 411 pr. para la subsecuencia común más larga, 390–397 para la subsecuencia palíndromo más larga, 405 pr. para el camino simple más largo en un gráfico acíclico dirigido ponderado, 404 pr. para multiplicación de cadenas de matrices, 370–378 y memorización, 387–389 para árboles de búsqueda binarios óptimos, 397–404 subestructura óptima en, 379–384 subproblemas superpuestos en, 384–386 para imprimir claramente, 405 pr. reconstrucción de una solución óptima en, 387 relación con divide y vencerás, 359 para cortar varillas, 360–370 para tallar costuras, 409 pr. para la firma de agentes libres, 411 pr. de arriba hacia abajo con memorización, 365 para cierre transitivo, 697–699 para algoritmo de Viterbi, 408 pr. para 0-1 problema de mochila, 427 ej. ver también estructura de datos tabla dinámica, 463–471 analizado por método contable, 465–466 analizado por análisis agregado, 465 analizado por método potencial, 466–471 factor de carga de, 463 árbol dinámico, 482
- e, 55 E CE (valor esperado), 1197 primera forma temprana, 444 tarea temprana, 444 borde, 1168 admisible, 749 antiparalelo, 711–712 atributos de, 592 espalda, 609 puente, 621
- cuerda cónicoparabólica, 778 capacidad de, 709 clasificación en búsqueda en amplitud, 621 pr. clasificación en búsqueda en profundidad, 609

- continuación, 778
- crítico, 729
- cruz, 609
- adelante, 609
- inadmisible, 749
- ligero, 626
- peso negativo, 645–646
- residual, 716
- retorno, 779
- seguro,
- 626 saturado,
- 739 desove,
- 778 árbol, 601, 603,
- 609 peso de,
- 591 conectividad de borde,
- 731 ex. juego de bordes, distancia de edición 1168, 406 pr.
- Algoritmo de Edmonds-Karp, 727–730
- evento elemental, 1189
- inserción elemental, 465
- elemento de un conjunto (2), 1158 algoritmo elipsoide, 850, 897
- método de factorización de curva elíptica, 984
- elseif, en pseudocódigo, 20 n.
- else, en pseudocódigo, 20
- lenguaje vacío (;), 1058
- conjunto vacío (;), 1158
- leyes de conjuntos vacíos, 1159 pila vacía, 233 cadena vacía ("), 986, 1058 árbol vacío, 1178 codificación de instancias de problemas, 1055–
  - 1057 punto final de un intervalo, 348 de un segmento de línea, 1015 ENQUEUE, 235 ingreso de un vértice, 1169 ingreso de variable, 867 función de entropía, 1187 gráfico denso, 706 pr.-función hash
    - universal, 269 ej., igualdad de funciones, 1166 lineal, 845 de conjuntos, 1158 restricción de igualdad, 670 ej., 852 y restricciones de desigualdad, 853 estrecha, 865 violación de, 865
- ecuación y notación asintótica, 49–50 normal, 837
- recurrencia, ver recurrencia
- clase de equivalencia, 1164
- módulo n ( $\mathbb{Z}_n$ ), 928
- equivalencia, modular (), 54, 1165 ex. relación de equivalencia, 1164 y equivalencia modular, 1165 ex.
- programas lineales equivalentes, 852
- error, en pseudocódigo, 22
- problema de escape, 760 pr.
- EUCLID, 935
- Algoritmo de Euclides, 933–939, 981 pr., 983
- Distancia euclidiana, 1039
- Norma euclidiana (kk), 1222
- Constante de Euler, 943
- Función phi de Euler, 943
- Teorema de Euler, 954, 975 ex.
- Tour de Euler, 623 pr., 1048
  - y ciclos hamiltonianos, 1048
- evaluación de un polinomio, 41 pr., 900, 905 ej. derivados de, 922 pr. en múltiples puntos, 923 pr.
- evento, 1190
- punto de evento, 1023 cronograma de punto de evento, 1023 SUMA DE
- SUBCONJUNTO
- EXACTO, 1129 exceso de flujo, 736 propiedad de intercambio, 437 exclusión e inclusión, 1163 ej. ejecutar una subrutina, 25 n. expansión de una tabla dinámica, 464–467 expectativa, véase valor esperado tiempo de ejecución esperado, 28, 117 valor esperado, 1197–1199 de una distribución binomial, 1204 de una distribución geométrica, 1202 de una variable aleatoria indicadora, 118 vértice explorado, 605 función exponencial, 55–56 altura exponencial, 300 árbol de búsqueda exponencial, 212, 483 serie exponencial, 1147 instrucción de exponentiación, 24 exponentiación, modular, 956 EXTENDED-BOTTOM-UP-CUT-ROD, 369

- EXTENDIDO-EUCLID, 937  
 EXTENDIDO-MÁS-CORTO-CAMINOS, 688  
 extensión de un conjunto,  
 438 exterior de un polígono, 1020 ej.  
 nodo externo, 1176  
 longitud de ruta externa, 1180 ex.  
 extrayendo la clave máxima de los  
     montones d-ary, 167 pr. de max-  
     heaps, 163 extrayendo la  
 clave mínima  
     de montones de Fibonacci, 512–518 de  
     2-3-4 montones, 529 pr. de  
         Young tableaus, 167 pr.  
 EXTRACTO-MAX, 162–163  
 EXTRACTO-MIN, 162, 505  
  
 factor, 928  
 giro, 912  
 función factorial ( $\tilde{S}$ ), 57–58  
 factorización, 975–980, 984 único,  
 931 falla, en  
 un ensayo de Bernoulli, 1201 moneda  
 justa, 1191 fan-  
 out, 1071 lema  
 de Farkas, 895 pr. problema  
 del par más lejano, 1030 RUTAS  
 MÁS RÁPIDAS PARA TODOS LOS PARES, 691, 692 ex.  
  
 transformada rápida de Fourier (FFT), 898–925  
     círculo para, 919–920  
     implementación iterativa de, 915–918  
     multidimensional, 921 pr.  
     algoritmo multiproceso para, 804 ex.  
     implementación recursiva de, 909–912 usando  
         aritmética modular, 923 pr.  
     problema de viabilidad, 665, 894 pr.  
     programa lineal factible, 851 región  
         factible, 847 solución  
         factible, 665, 846, 851 teorema de  
             Fermat, 954 FFT, ver  
     transformada rápida de Fourier FFTW,  
     924 FIB, 775  
 FIB-HEAP-  
 CHANGE-KEY, 529 pr.  
 FIB-HEAP-DECREASE-KEY, 519 FIB-HEAP-  
 DELETE, 522 FIB-HEAP-  
 EXTRACT-MIN, 513 FIB-HEAP-INSERT,  
 510  
 FIB-HEAP-LINK, 516 FIB-  
 HEAP-PRUNE, 529 pr.  
 FIB-HEAP-UNION, 512 Montón  
 de Fibonacci, 505–530  
     cambio de llave, 529 pr. en  
     comparación con montones binarios, 506–507  
     creación, 510  
     disminución de una entrada, 519–522  
     eliminación de, 522, 526 pr. en el  
         algoritmo de Dijkstra, 662 extracción  
         de la clave mínima de, 512–518 inserción en, 510–511  
         en el algoritmo de Johnson,  
         704 grado máximo de, 509, 523–  
         526 clave mínima de, 511 función potencial  
         para, 509 en el algoritmo  
         de Prim, 636 poda , 529 pág.  
         tiempos de ejecución de las  
         operaciones en, 506  
         fig. uniendo, 511–512 números de Fibonacci, 59–  
         60, 108 pr., 523  
         cálculo de, 774–780, 981 pr.  
  
 FIFO (primero en entrar, primero en salir),  
 232 véase también  
 función de estado final de cola,  
 996 hebra final, 779  
 ENCONTRAR-PROFOUNDIDAD, 583 pr.  
 FIND-MAX-CROSSING-SUBARRAY, 71 FIND-  
 MAXIMUM-SUBARRAY, 72 find path, 569  
 FIND-SET, 562  
 implementación  
     disjoint-set-forest de, 571,  
         585  
     implementación de lista enlazada de, 564  
 vértice terminado, 603  
 tiempo de finalización, búsqueda en profundidad primero, 605  
     y componentes fuertemente conectados, 618  
 tiempo de finalización, en selección de  
 actividad, 415 autómata  
     finito, 995 para coincidencia de  
         cadenas, 996–1002 FINITE-AUTOMATON-  
         MATCHER, 999  
     grupo finito, 940 secuencia  
         finita, 1166  
     conjunto finito, 1161 heuristic  
         de primer ajuste, 1134 pr.  
         primero en entrar,  
         primero en salir, 232 ver también cola de código de longitud fija, 429

- tipo de datos de coma flotante, 23
- función de piso (bc), 54 en
  - teorema maestro, 103–106
- instrucción de piso, 23
- flujo, 709–714
  - agregado, 863
  - aumento de, 716 bloqueo, 765 cancelación de, 717 exceso, 736 valor entero, 733 neto, a través de un corte, 720 valor de, 710 conservación de flujo, 709–710 red de flujo, 709–714 correspondiente a un gráfico bipartito, 732 corte de, 720–724 con múltiples fuentes y sumideros, 712
- FLOYD-WARSHALL, 695
- FLOYD-WARSHALL0 , 699 ej.
- Algoritmo de Floyd-Warshall, 693–697, 699–700 ej., 706 multiproceso, 797 ej.
- FORD-FULKERSON, 724 Ford-Fulkerson method, 714–731, 765 FORD-FULKERSON-METHOD, 715 forest, 1172–1173 depth-first, 603 disjoint-set, 568–572 for, in pseudocode, 20–21 and loop invariants , 19 n. serie de potencia formal, 108 pr. satisfacibilidad de la fórmula, 1079–1081, 1105 arista anterior, 609 sustitución directa, 816–817 Transformada de Fourier, véase transformada de Fourier discreta, transformada rápida de Fourier, problema fraccional de la mochila, 426, 428 ej. agente libre, 411 pr. liberación de objetos, 243–244 lista libre, 243 OBJETO LIBRE, 244 árbol libre, 1172–1176 dominio de frecuencia, 898 árbol binario completo, 1178, 1180 ex. relación con el código óptimo, 430 nodo completo, 489 rango completo, 1223 caminata completa de un árbol, 1114 producto de cadena de matriz completamente entre paréntesis, 370 esquema de aproximación de tiempo polinomial completo, 1107 para suma de subconjuntos, 1128–1134, 1139 función, 1166–1168 de Ackermann, 585 base, 835 convexo, 1199 estado final, 996 hash, véase función hash lineal, 26, 845 objetivo, 664, 847, 851 potencial, 459 prefijo, 1003–1004 cuadrático, 27 reducción, 1067 sufijo, 996 transición, 995, 1001–1002, 1012 ej. iteración funcional, 58 teorema fundamental de la programación lineal, 892 estrategia más lejana en el futuro, 449 pr. árbol de fusión, 212, 483 clasificación difusa, 189 pr.
- Algoritmo de escala de Gabow para rutas más cortas de fuente única, 679 pr. carácter de espacio, 989 ej., 1002 ej. brecha heurística, 760 ej., 766 recolección de basura, 151, 243 compuerta, 1070 Eliminación gaussiana, 819, 842 mcd, ver el máximo común divisor general tamiz de campo numérico, 984 función generadora, 108 pr. generador de un subgrupo, 944 de Z nota, 955 GENÉRICO-MST, 626 GENERIC-PUSH-RELABEL, 741 algoritmo genérico push-relabel, 740–748 distribución geométrica, 1202–1203 y bolas y contenedores, 134 series geométricas, 1147 geometría, computacional, 1014–1047 GF.2/, 1227 pr. envoltorio de regalo, 1037, 1047

- variable global, 21  
 algoritmo de Goldberg, véase proporción áurea  
     del algoritmo  
 push-relabel (), 59, 108 pr. chismes,  
 478 INJERTO,  
 583 pr.  
 Escaneo de Graham, 1030–1036, 1047  
 GRAHAM-SCAN, gráfico  
 1031, 1168–1173  
     representación de lista de adyacencia de, 590  
     representación de matriz de adyacencia de, 591  
     algoritmos para, 587–766 y  
     notación asintótica, 588 atributos de,  
 588, 592 búsqueda en  
     amplitud de, 594–602, 623 coloración de, 1103  
     pr. complemento de, 1090  
     componente, 617 restricción,  
 666–668 densa, 589  
     búsqueda en profundidad  
     de, 603–612,  
 623 dinámica, 562 n. -denso, 706 pr.  
 hamiltoniano, 1061  
     matriz de incidencia  
     de, 448 pr., 593 ex.  
 intervalo, 422 ej. no hamiltoniano, 1061  
     camino más corto  
     en, 597 conectados  
     individualmente, 612 ex.  
 escaso, 589 estático, 562 n.  
     subproblema,  
 367–368  
     recorrido por, 1096  
 ponderado, 591  
     véase también  
     gráfico acíclico dirigido, gráfico dirigido, red de flujo,  
         gráfico no dirigido,  
         árbol  
 matriz gráfica, 437–438, 642 GRÁFICO-  
 ISOMORFISMO, 1065 ej. vértice gris, 594, 603  
     máximo común divisor  
     (mcd), 929–930, 933 ex.  
 algoritmo mcd binario para, 981 pr.  
 Algoritmo de Euclides para, 933–939, 981 pr., 983 con  
     más de dos argumentos, 939 ej. teorema de  
     recursión para, 934 greedoid, 450  
 GREEDY, 440  
 SELECTOR DE ACTIVIDAD CODICIOSO, 421  
 algoritmo codicioso, 414–450  
     para selección de actividad, 415–422  
     para cambio de moneda, 446 pr.  
     en comparación con la programación dinámica, 381,  
         390 ej., 418, 423–427  
 Algoritmo de Dijkstra, 658–664  
     elementos de, 423–428  
     para el problema de la mochila fraccional, 426  
 Propiedad de elección codiciosa en, 424–425  
     para el código de Huffman, 428–  
 437 Algoritmo de Kruskal, 631–633 y  
     matroids, 437–443 para árbol  
     de expansión mínimo, 631–638 para  
     programación de subprocessos múltiples, 781–783  
     para almacenamiento en caché  
     fuera de línea, 449 pr. subestructura  
     óptima en, 425 Algoritmo de Prim,  
 634–636 para establecer cobertura,  
 1117–1122, 1139 para programación de tareas, 443–  
 446, 447–448 pr. en un matroide  
     ponderado, 439–442 para cubierta de  
 conjunto ponderado, 1135 pr. propiedad  
     de elección codiciosa, 424–425 de  
     selección de actividad, 417–418 de  
     los códigos Huffman, 433–434 de  
 una matroide ponderada,  
 441 programador codicioso, 782  
 GREEDY-SET-  
 COVER, cuadrícula  
 1119, 760 pr.  
     grupo, 939–946 cílico,  
 955 operador ('), 939 adivinar la solución, en el  
     método de sustitución, 84–85  
     mitad 3-CNF satisfacibilidad, 1101 ex.  
 intervalo semiabierto, 348  
 Teorema de Hall, 735 ej.  
 problema de detención, 1048  
 lema de la mitad, 908  
 HAM-CYCLE, 1062 ciclo  
 hamiltoniano, 1049, 1061, 1091–1096, 1105  
     gráfico hamiltoniano, 1061  
     camino hamiltoniano, 1066 ej., 1101 ej.  
 HAM-PATH, 1066 ex. mango,  
 163, 507 lema de  
     apretón de manos, 1172 ex.

- número armónico, 1147, 1153–1154 serie armónica, 1147, 1153–1154 HASH-DELETE, 277 ej. función hash, 256, 262–269 auxiliar, 272 resistente a colisiones, 964 método de división para, 263, 268–269 ex. -universal, 269 ej. método de multiplicación para, 263–264 universal, 265–268 hashing, 253–285 con encadenamiento, 257–260, 283 pr. doble, 272–274, 277 ej. k-universal, 284 pr. en memorización, 365, 387 con direccionamiento abierto, 269–277 perfecto, 277–282, 285 para reemplazar listas de adyacencia, 593 ej. universal, 265–268 HASH-INSERT, 270, 277 ex. HASH-SEARCH, 271, 277 ej. tabla hash, 256–261 dinámica, 471 ex. secundario, 278 véase también hash valor hash, 256 problema de control de sombreros, 122 ej. cabeza en una unidad de disco, 485 de una lista enlazada, 236 de una cola, 234 montón, 151–169 analizado por método potencial, 462 ej. binomial, 527 pr. edificio, 156–159, 166 pr. en comparación con montones de Fibonacci, 506–507 d-ary, 167 pr., 706 pr. supresión de, 166 ej. en el algoritmo de Dijkstra, 662 extrayendo la clave máxima de, 163 Fibonacci, vea el montón de Fibonacci como almacenamiento recolectado de basura, 151 altura de, 153 en el algoritmo de Huffman, 433 para implementar un montón fusionable, 506 aumentando una clave en, 163–164 inserción en , 164 en el algoritmo de Johnson, 504 max-heap, 152 clave máxima de, 163 fusionable, ver montón fusionable min-heap, 153 en el algoritmo de Prim, 636 como cola de prioridad, 162–166 relajado, 530 tiempos de ejecución de operaciones en, 506 fig. y tratos, 333 pr. 2-3-4, 529 pr. HEAP-DECREASE-KEY, 165 ej. HEAP-DELETE, 166 ej. MONTÓN-EXTRACTO-MAX, 163 MONTÓN-EXTRACTO-MIN, 165 ej. HEAP-INCREASE-KEY, 164 HEAP-MAXIMO, 163 HEAP-MINIMUM, 165 ej. propiedad de montón, 152 mantenimiento de, 154–156 frente a propiedad de árbol de búsqueda binaria, 289 ej. heapsort, 151–169 HEAPSORT, 160 talón, 602 ej. altura de un árbol binomial, 527 pr. negro-, 309 de un árbol B, 489–490 de un montón d-ario, 167 pr. de un árbol de decisión, 193 exponencial, 300 de un montón, 153 de un nodo en un montón, 153, 159 ex. de un nudo en un árbol, 1177 de un árbol rojo-negro, 309 de un árbol, 1177 árbol de altura equilibrada, 333 pr. función de altura, en algoritmos push-relabel, 738 familia hereditaria de subconjuntos, 437 matriz hermética, 832 ex. punto final alto de un intervalo, 348 función alta, 537, 546 ASISTENTE DE CONTRATACIÓN, 115 problema de contratación, 114–115, 123–124, 145 en línea, 139–141 análisis probabilístico de, 120–121 éxito caché, 449 pr. memoria, 991

- HOARE-TABIQUE, 185 pr.  
 HOPCROFT-KARP, 764 pr.  
 Algoritmo de emparejamiento bipartito Hopcroft-Karp,  
     763 pr.  
 rayo horizontal, 1021 ej.  
 Regla de Horner, 41 pr., 900  
     en el algoritmo Rabin-Karp, 990  
 HUFFMAN, 431  
 código Huffman, 428–437, casco  
 450, convexo, 8, 1029–1039, 1046 pr.  
 Proyecto Genoma Humano, 6  
 hiperborde, 1172  
 hipergrafía, 1172 y  
     gráficos bipartitos, 1173 ex.
- computadora paralela ideal, 779  
 leyes de idempotencia para conjuntos,  
 1159 identidad,  
 939 matriz identidad,  
 1218 si, en pseudocódigo,  
 20 imagen,  
 1167 compresión de imagen, 409 pr.,  
 413 borde inadmisible, 749  
 incidencia, 1169  
 matriz de incidencia  
     y restricciones de diferencia, 666 de  
     un gráfico dirigido, 448 pr., 593 ex. de un  
     grafo no dirigido, 448 pr. inclusión y  
     exclusión, 1163 ex. paso incompleto,  
 782 INCREASE-KEY,  
 162 aumentar una clave,  
 en un montón máximo, 163–164 INCREMENT,  
 454 método de  
 diseño incremental, 29  
     para encontrar el casco convexo, 1030  
 en grado, 1169  
 sangría en pseudocódigo, 20  
 independencia  
     de eventos, 1192–1193, 1195 ej. de  
     variables aleatorias, 1197 de  
     subproblemas en programación dinámica, 383–  
     384  
 familia independiente de subconjuntos,  
 437 conjunto independiente,  
     1101 pr. de  
 tareas, 444 hilos  
 independientes, 789 función  
 de índice, 537, 546 índice de un none elemento de Z
- indicador de variable aleatoria, 118–121  
 en el análisis de la altura esperada de un árbol de  
     búsqueda binario construido  
     aleatoriamente, 300–303 en el análisis de la  
     inserción en un treap, 333 pr. en  
 análisis de rayas, 138–139 en análisis de la paradoja  
 del cumpleaños, 132–133 en algoritmo de aproximación para  
     Satisfacción de MAX-3-CNF, 1124 al  
 delimitar la cola derecha de la distribución  
     binomial, 1212–1213 en el  
 análisis de clasificación de cubos, 202–  
 204 valor esperado de,  
 118 en el análisis hash, 259–260  
 en el análisis del problema de contratación,  
 120–121 y linealidad de la expectativa,  
 119 en análisis quicksort, 182–184, 187 pr.  
 en análisis de selección aleatoria, 217–219,  
 226 pr.  
 en análisis de hashing universal, 265–266  
 subgrafo inducido, 1171  
 restricción de desigualdad,  
     852 y restricciones de igualdad,  
 853 desigualdad, lineal,  
 846 programa lineal no factible, 851  
 solución no factible, 851  
 secuencia infinita, 1166  
 conjunto infinito,  
 1161 suma infinita,  
 1145 infinito, aritmética con, 650  
 INITIALIZE-PRFLOW, 740  
 INITIALIZE-SIMPLEX, 871, 887  
 INITIALIZE-SINGLE-SOURCE, 648 hilo  
 inicial, 779 función  
 inyectiva, 1167 producto  
 interno, 1222 recorrido  
 de árbol en orden, 287, 293 ex., 342  
 INORDER-TREE-WALK, 288  
 clasificación en el lugar, 17, 148, 206  
 pr.  
     entrada a un  
     algoritmo, 5 a un circuito combinacional,  
 1071 distribución de, 116,  
 122 a una puerta lógica,  
 1070  
 tamaño de, 25 alfabeto  
 de entrada, 995 INSERTAR, 162,  
 230, 463  
     ej., 505 inserción en árboles binarios de búsqueda, 294 –295

- en un vector de bits con un árbol binario superpuesto, 534 en un
  - vector de bits con un árbol superpuesto de altura constante, 534 en árboles
- B, 493–497 en tablas hash
- encadenadas, 258 en montones d-arios, 167 pr. en tablas de direcciones directas, 254 en tablas dinámicas, 464–467 elementales, 465 en montones de Fibonacci, 510–511 en montones, 164 en árboles de intervalo, 349 en listas enlazadas, 237–238 en tablas hash de direcciones abiertas, 270 en orden -árboles estadísticos, 343 en estructuras proto van Emde Boas, 544 en colas, 234 en árboles rojo-negro, 315–323 en pilas, 232 en estados de línea de barrido, 1024 en trampas, 333 pr. en 2-3-4 montones, 529 pr. en los árboles de van Emde Boas, 552–554 en los cuadros de Young, 167 pr. clasificación por inserción, 12, 16–20, 25–27 en clasificación por cubo, 201–204 en comparación con clasificación por fusión, 14 ej. en comparación con quicksort, 178 ex. árbol de decisión para, 192 fig. en ordenación por fusión, 39 pr. en clasificación rápida, 185 ej. usando búsqueda binaria, 39 ej.
- CLASIFICACIÓN POR INSERCIÓN, 18, 26, 208 pr. instancia
  - de un problema abstracto, 1051, 1054 de un problema, 5
- instrucciones del modelo RAM, 23 tipo de datos enteros, 23 programación lineal entera, 850, 895 pr., 1101 ej.
- enteros ( $\mathbb{Z}$ ), 1158 flujo de valores enteros, 733 teorema de integralidad, 734 integral, sumas aproximadas, 1154–1156 integración de una serie, 1147 interior de un polígono, 1020 ej.
- método de punto interior, 850, 897 vértice intermedio, 693 nodo interno, 1176 longitud de ruta interna, 1180 ex. interpolación por spline cúbico, 840 pr. interpolación por un polinomio, 901, 906 ej. en raíces complejas de unidad, intersección 912–913
- de acordes, 345 ej.
- determinando, para un conjunto de segmentos de línea, 1021–1029, 1047 determinando, para dos segmentos de línea, 1017–1019 de idiomas, 1058 de conjuntos ( $\mathcal{I}$ ), 1159 intervalo, 348
  - clasificación difusa de, 189 pr.
  - INTERVALO-DELETE, gráfico de 349 intervalos, 422 ej.
  - INTERVALO-INSERTAR, 349
  - INTERVALO-BÚSQUEDA, 349, 351
  - INTERVALO-BÚSQUEDA-EXACTAMENTE, 354 ej. árbol de intervalo, 348–354 tricotomía de intervalo, 348 intratabilidad, 1048 turno no válido, 985 planificación de inventario, 411 pr. inverso
    - de una función biyectiva, 1167 en un grupo, 940 de una matriz, 827–831, 842, 1223, 1225 ej. multiplicativo, módulo n, inversión 949
    - en una lista autoorganizada, 476 pr. en una secuencia, 41 pr., 122 ej., 345 ej. inversor, 1070 matriz invertible, 1223 vértice aislado, 1169 gráficos isomorfos, 1171 función iterada, 63 pr. función de logaritmo iterado, 58–59 ITERATIVE-FFT, 917 ITERATIVE-TREE-SEARCH, 291 iter function, 577
  - Marcha de Jarvis, 1037–1038, 1047 Desigualdad de Jensen, 1199 JOHNSON, 704

- Algoritmo de Johnson, 700–706  
 unión  
   de árboles rojo-negros, 332  
   pr. de 2-3-4 árboles, 503  
   pr. función de densidad de probabilidad conjunta, 1197 permutación de Josefo, 355 pr.
- Algoritmo de Karmarkar, 897  
 Algoritmo de ciclo de peso medio mínimo de Karp, 680  
   pr.  
 árbol k-ario, 1179  
 k-CNF, 1049 k-  
   coloring, 1103 pr., 1180 pr. k-  
   combinación, 1185 k-  
   forma normal conjuntiva, 1049 núcleo de un polígono, 1038 ex. clave, 16, 147, 162, 229 ficticia, 397 interpretada como un número natural, 263 mediana, de un nodo de árbol B, 493 pública, 959, 962 secreta, 959, 962 estática, 277 palabras clave, en pseudocódigo, 20–22 multiproceso, 774, 776–777, 785–786 “adversario asesino” para clasificación rápida, 190 ley actual de Kirchhoff, 708 estrella de Kleene (), 1058 algoritmo KMP, 1002–1013 KMP-MATCHER, 1005 problema de la mochila fraccional, 426, 428 ej. 0-1, 425, 427 ej., 1137 pr., 1139 k-árbol vecino, 338 nudo, de una spline, 840 pr. Algoritmo de Knuth-Morris-Pratt, 1002–1013 k-permutación, 126, 1184 Desigualdad de Kraft, 1180 ej. Algoritmo de Kruskal, 631–633, 642 con pesos de borde enteros, 637 ej. k-ordenado, 207 pr. k-cadena, 1184 k-subconjunto, 1161 k-subcadena, 1184 k-ésima potencia, 933 ex. hash k-universal, 284 pr. fórmula de Lagrange, 902 Teorema de Lagrange, 944 Teorema de Lamé, 936 lenguaje, 1057 completilidad de, 1077 ex. probando NP-completo de, 1078–1079 verificación de, 1063 último en entrar, primero en salir, 232 ver también pila tarea tardía, 444 capas convexas, 1044 pr. máximo, 1045 pr. ACV, 584 pr. mcm (mínimo común múltiplo), 939 ej. LCS, 7, 390–397, 413 LCS-LENGTH, 394 submatriz principal, 833, 839 ex. hoja, 1176 antepasado menos común, 584 pr. mínimo común múltiplo, 939 ej. aproximación de mínimos cuadrados, 835–839 dejando un vértice, 1169 dejando una variable, 867 IZQUIERDA, 152 hijo izquierdo, 1178 hijo izquierdo, representación del hermano derecho, 246, 249 ej. GIRO A LA IZQUIERDA, 313, 353 ex. rotación izquierda, 312 columna izquierda, 333 pr. subárbol izquierdo, 1178 Símbolo de Legendre  $\frac{a}{p}$ , 982 pr. longitud de un camino, 1170 de una secuencia, 1166 de una columna vertebral, 333 pr. de una cuerda, 986, 1184 nivel de una función, 573 de un árbol, 1177 función de nivel, 576 lexicográficamente menor que, 304 pr. clasificación lexicográfica, 304 pr. lg (logaritmo binario), 56 lg (función de logaritmo iterado), 58–59 lgk (exponenciación de logaritmos), 56 lg lg (composición de logaritmos), 56 LIFO (último en entrar, primero en salir), 232

- ver también pila
- borde ligero, 626
- restricción lineal, 846
- dependencia lineal, 1223
- igualdad lineal, 845
- resolución de
  - ecuaciones lineales modulares, 946–950 resolución de sistemas de, 813–827 resolución de sistemas tridiagonales de, 840 pr.
  - función lineal, 26, 845
  - independencia lineal, 1223 desigualdad lineal, 846 problema de viabilidad de desigualdad lineal, 894 pr.
    - linealidad de expectativa, 1198 e indicador de variables aleatorias, 119 linealidad de sumas, 1146
    - orden lineal, 1165 permutación lineal, 1229 pr. sondeo lineal, 272 programación lineal, 7, 843–897 algoritmos para, 850 aplicaciones de, 849 dualidad en, 879–886 algoritmo elíptico para, 850, 897 encontrar una solución inicial en, 886–891 teorema fundamental de, 892 punto interior métodos para, 850, 897 Algoritmo de Karmarkar para, 897 y flujo máximo, 860–861 y circulación de costo mínimo, 896 pr. y flujo de costo mínimo, 861–862 y flujo de múltiples productos básicos de costo mínimo, 864 ej.
    - y flujo de productos múltiples, 862–863 algoritmo simplex para, 864–879, 896 y ruta más corta de un solo par, 859–860 y rutas más cortas de fuente única, 664–670, 863 ej.
    - forma holgada para, 854–857 forma estándar para, 850–854 véase también programación lineal entera, 0–1 programación de enteros
    - relajación de programación lineal, 1125
    - búsqueda lineal, 22 ej.
    - aceleración lineal,
    - segmento de línea
      - 780, 1015 comparable, 1022 giro determinante de, 1017
  - determinar si alguno se cruza, 1021–1029, 1047
  - determinar si dos se intersecan, 1017–1019
  - enlace
    - de árboles binomiales, 527 pr.
    - de raíces del montón de Fibonacci, 513 de árboles en un bosque inconexo, 570–571 LINK,
    - 571 lista enlazada, 236–241 compacto, 245 ej., 250 pr. eliminación de, 238 para implementar conjuntos disjuntos, 564–568 inserción en, 237–238 lista de vecinos, 750 búsqueda, 237, 268 ej. autoorganizado, 476 pr.
    - lista, ver lista enlazada
  - LIST-DELETE, 238
  - LIST-DELETEO , 238
  - LIST-INSERT, 238
  - LIST-INSERTO , 240
  - BÚSQUEDA EN LISTA, 237 BÚSQUEDA , 239
  - EN LISTA0
    - literal, 1082 notación oh pequeña, 50–51, 64 notación omega pequeña, 51 distancia Lm, 1044 ej. In (logaritmo natural), 56 factor de carga
      - de una tabla dinámica, 463
      - de una tabla hash, 258
    - instrucción de carga, 23 variable local, 21
    - función logarítmica (log), 56–57
      - discreta, 955
      - iterada (lg), 58–59
    - paralelismo lógico, 777
    - puerta lógica, 1070
    - subsecuencia común más larga, 7, 390–397, 413
    - subsecuencia palíndromo más larga , 405 pr.
  - RTA MÁS LARGA, 1060 ej.
  - LONGITUD DE LA RUTA MÁS LARGA, 1060 ej. ciclo simple más largo, 1101 ej.
  - camino simple más largo, 1048
    - en un gráfico no ponderado, 382
    - en un gráfico acíclico dirigido ponderado, 404 pr.
  - CADENA DE BÚSQUEDA, 388

- bucle, en pseudocódigo, 20
- paralelo, 785–787
- bucle invariable, 18–19
  - para búsqueda en amplitud, 595
  - para construir un montón,
  - 157 para consolidar la lista raíz de un montón de Fibonacci,
  - 517 para determinar el rango de un elemento en un árbol estadístico de orden,
  - 342 para el algoritmo de Dijkstra,
  - 660 y para bucles, 19
  - n. para el método de árbol de expansión mínimo genérico, 625
  - para el algoritmo genérico push-relabel, 743 para el escaneo de Graham, 1034
  - para heapsort, 160 ej.
  - para la regla de Horner, 41
  - pr. para aumentar una clave en un montón, 166 ej. inicialización
  - de, 19 para clasificación
  - por inserción, 18
  - mantenimiento de,
  - 19 para fusión, 32 para exponentiación
  - modular, 957
  - origen de, 42 para
  - partición, 171 para el algoritmo
  - de Prim, 636 para el algoritmo de Rabin-Karp, 993 para permutar aleatoriamente una matriz, 127 , 128 ej.
  - para la inserción de árboles rojo-negro, 318 para el algoritmo de reetiquetado al frente, 755 para buscar un árbol de intervalos, 352 para el algoritmo simplex, 872
  - para autómatas de coincidencia de cadenas, 998, 1000 y terminación, 19
  - punto final inferior de un intervalo, 348
  - límites inferiores
    - en aproximaciones, 1140
    - asintóticas, 48
    - para clasificación promedio, 207
    - pr. sobre coeficientes binomiales, 1186 para comparar jarras de agua, 206
    - pr. para casco convexo, 1038 ej., 1047 para estructuras de datos de conjuntos disjuntos, 585 para encontrar el mínimo, 214 para encontrar el predecesor, 560 para la duración de un recorrido óptimo de vendedor ambulante, 1112–1115
  - para hallazgo mediano, 227
  - para fusión, 208 pr.
  - para cubierta de vértice de peso mínimo, 1124–1126
  - para cálculos de subprocessos múltiples, 780
  - y funciones potenciales, 478 para
  - operaciones de cola de prioridad, 531 y
  - recurrencias, 67 para
  - mínimo y máximo simultáneos, 215 ej.
  - para el tamaño de una cubierta de vértice óptima, 1110, 1135 pr. para clasificación, 191–194, 205 pr., 211, 531 para rayas, 136–138, 142 ej. en sumatorias, 1152, 1154
  - mediana inferior, 213
    - 546
  - raíz cuadrada inferior p#, matriz triangular inferior, 1219, 1222 ej., 1225 ej. función
  - baja, 537, 546
  - descomposición LU, 806 pr., 819–822
  - DESCOMPOSICIÓN LU**, 821
  - descomposición LUP, 806 pr., 815
    - cálculo de, 822–825 de una matriz diagonal, 827 ej. en
    - inversión de matrices, 828 y
    - multiplicación de matrices, 832 ej. de una matriz de permutación, 827 ej. uso de, 815–819 LUP-
  - DESCOMPOSICIÓN**, 824 LUP-
  - SOLVE**, 817
  - memoria principal, 484
  - MAKE-HEAP**, 505
  - MAKE-SET**, 561
    - implementación de bosque de conjunto disjunto de, 571 implementación de lista enlazada
  - de, 564makespan, 1136 pr.
  - HACER-ÁRBOL**, 583 pr.
  - Distancia de Manhattan, 225 pr., 1044 ej.
  - nodo marcado, 508, 519–520
  - Desigualdad de Markov, 1201 ej.
  - método maestro para resolver una recurrencia, 93–97
  - teorema maestro, 94
    - prueba de, 97–106
  - vértice coincidente, 732
  - bipartito
  - coincidente, 732, 763 pr.

- máximo, 1110, 1135 pr. máximo,  
1135 pr. y flujo máximo,  
732–736, 747 ej. perfecto, 735 ej. de cadenas,  
985–1013 bipartito  
ponderado, 530 matriz  
matroide, 437 matriz, 1217–  
1229 adición de, 1220  
adyacencia, 591
  - transposición conjugada  
de, 832 ej.
  - determinante de, 1224–1225 diagonal,  
1218 Hermitian, 832 ej. identidad,  
1218 incidencia,  
448 pr., 593 ej. inversión  
de, 806 pr., 827–  
831, 842 triangular inferior, 1219,  
1222 ej., 1225 ej. multiplicación de, véase  
multiplicación de matriz negativa de, 1220 permutación,  
1220, 1222 ex. predecesor, 685 producto de, con un  
vector, 785–787, 789–  
790,
  - 792 ej.
  - pseudoinversa de, 837
  - múltiplo escalar de, 1220 resta  
de, 1221 simétrica, 1220
  - simétrica definida  
positiva, 832–835, 842 Toeplitz, 921 pr. transposición  
de, 797 ej., 1217
  - transposición de, multiproceso, 792
  - ej. tridiagonal, 1219 unidad triangular inferior,  
1219 unidad triangular  
superior, 1219 triangular superior,  
1219, 1225 ex.
  - Vandermonde, 902, 1226 pr.
  - multiplicación matriz-cadena, 370–378 MATRIX-CHAIN-MULTIPLY  
MATRIZ-CADENA-ORDEN, 375
  - multiplicación de matrices, 75–83, 1221
    - para rutas más cortas de todos los pares, 686–  
693, 706–707
    - booleano, 832 ej. y  
calcular el determinante, 832 ej. método divide y  
vencerás para, 76–83 y descomposición LUP, 832  
ej. e inversión de matriz, 828–831, 842
  - algoritmo multihilo para, 792–797, 806 pr.
  - Método de Pan para, 82 ej.
  - Algoritmo de Strassen para, 79–83, 111–112 MATRIX-MULTIPLY, 371 multiplicación  
matriz-vector, multiproceso, 785–787, 792 ej. con carrera,  
789–790 matroid, 437–  
443, 448 pr., 450, 642
  - para programación de tareas, 443–446 MATRIX-VEC, 785 MAT-VEC-MAIN-LOOP, 786  
MAT-VEC-WRONG,  
790 MAX-CNF satisfacibilidad, 1127 ej.
  - Problema MAX-CUT, 1127 ej.
  - MAX-FLUJO POR ESCALADO, 763 pr. teorema  
de corte mínimo de flujo máximo, 723
  - montón máximo,  
152 edificio, 156–159 d-  
ario, 167 pr.  
supresión de, 166 ej.  
extrayendo la clave máxima de, 163 en heapsort,  
159–162 aumentando una  
clave en, 163–164 inserción en, 164  
clave máxima de, 163  
como una cola de máxima  
prioridad, 162–166 fusionable, 250 n., 481  
n., 505 norte.
  - MAX-HEAPIFY, 154 MAX-HEAP-INSERT, 164
    - construyendo un montón con, 166 pr.
    - propiedad max-heap, 152  
mantenimiento de, 154–156
  - elemento máximo, de un conjunto parcialmente ordenado,  
1165
  - capas máximas, 1045 pr.
  - coincidencia máxima, 1110, 1135 pr. punto  
máximo, 1045 pr. subconjunto  
máximo, en un matroide, 438 programa lineal  
de maximización, 846
    - y programas lineales de minimización, 852 máximo,  
213 en árboles
    - binarios de búsqueda, 291 de una  
distribución binomial, 1207 ej. en un vector de  
bits con un árbol binario superpuesto, 533 en un vector  
de bits con  
un árbol superpuesto de altura constante, 535

- hallazgo, 214–215  
 en montones,  
 163 en árboles de estadísticas de orden, 347 ej. en estructuras proto van Emde Boas, 544 ej. en árboles rojo-negros, 311 en árboles van Emde Boas, 550  
**MÁXIMO**, 162–163, 230  
 coincidencia bipartita máxima, 732–736, 747 ex., 766 Algoritmo Hopcroft-Karp para, 763 pr. grado máximo, en un montón de Fibonacci, 509, 523–526  
 caudal máximo, 708–766  
     Algoritmo de Edmonds-Karp para, 727–730  
     Método Ford-Fulkerson para, 714–731, 765 como programa lineal, 860–861 y  
     coincidencia bipartita máxima, 732–736, 747 ej.  
     algoritmos de reetiquetado push para, 736–760, 765 algoritmo de reetiquetado al frente para, 748–760 algoritmo de escalado para, 762 pr., 765 actualización, 762 pr. coincidencia máxima, 1135  
     pr. árbol de expansión máximo, 1137 pr.  
     problema de subarreglo máximo, 68–75, 111 cola de máxima prioridad, 162  
     satisfacibilidad de MAX-3-CNF, 1123–1124, 1139  
**MAYBE-MST-A**, 641 pr.  
**QUIZÁS-MST-B**, 641 pr.  
**QUIZÁS-MST-C**, 641 pr.  
 media, véase valor esperado  
 peso medio de un ciclo, 680 pr.  
 mediana, 213–227  
     algoritmo multiproceso para, 805 ex. de listas ordenadas, 223 ej.  
     de dos listas ordenadas, 804 ej.  
     ponderado, 225 pr.  
 clave mediana, de un nodo de árbol B, 493 método mediana de 3, 188 pr.  
 miembro de un conjunto (2), 1158 membresía  
     en estructuras proto van Emde Boas, 540–541 en árboles Van Emde Boas, 550  
 memorización, 365, 387–389  
**MEMOIZED-CUT-ROD**, 365  
**MEMOIZED-CUT-ROD-AUX**, 366  
**CADENA-MATRIZ-MEMOIZADA**, 388  
 memoria, 484  
 jerarquía de memoria, 24  
**MERGE**, 31  
 montón fusionable, 481, 505  
     montones binomiales, 527  
     pr. implementación de lista enlazada de, 250  
     pr. montones relajados, 530 tiempos de ejecución de operaciones en, 506 fig. 2-3-4 montones, 529 pr. véase también Fibonacci heap mergeable max-heap, 250 n., 481 n., 505 n. min-heap fusionable, 250 n., 481 n., 505 MERGE-LISTS, 1129  
 merge sort, 12, 30–37 en  
     comparación con la ordenación por inserción, 14 ej. algoritmo multiproceso para, 797–805, 812 uso de ordenación por inserción en, 39 pr.  
**MERGE-SORT**, 34  
**MERGE-SORT0**, 797  
 fusión de  
     k listas ordenadas, 166 ej.  
     límites inferiores para, 208 pr.  
     algoritmo multiproceso para, 798–801 de dos matrices ordenadas, 30  
**MILLER-RABIN**, 970  
 Prueba de primalidad de Miller-Rabin, 968–975, 983 MIN-GAP, 354 ej.  
 min-heap, 153  
     analizados por método potencial, 462 ex.  
     edificio, 156–159 d-ario, 706 pr. en  
     el algoritmo de Dijkstra, 662 en el algoritmo de Huffman, 433 en el algoritmo de Johnson, 704  
     fusionables, 250 n., 481 n., 505  
     como cola de prioridad mínima, 165  
     ex. en el algoritmo de Prim, 636  
**MIN-HEAPIFY**, 156 ej.  
**INSERCIÓN MIN-HEAP**, 165 ex.  
 ordenamiento de min-heap, 507 propiedad de min-heap, 153, 507 mantenimiento de, 156  
     ex. en tratos, 333  
     pr. vs. propiedad de árbol de búsqueda binaria, 289 ej.  
 programa lineal de minimización, 846 y  
     programas lineales de maximización, 852  
 mínimo, 213 en  
     árboles binarios de búsqueda, 291

- en un vector de bits con un árbol binario
- superpuesto, 533 en un vector de bits con un árbol
  - superpuesto de altura
- constante, 535 en
- árboles B, 497 ej. en
- montones de
- Fibonacci, 511
- hallazgos, 214–215 fuera de línea,
- 582 pr. en árboles estadísticos de orden, 347 ej. en estructuras proto van Emde Boas, 541–542 en
- árboles rojo-negros, 311 en
- montones 2-3-4, 529 pr. en van Emde Boas árboles, 550 MÍNIMO, 162, 214, 230, 505 circulación de costo
- mínimo, 896 pr. flujo de costo mínimo, 861–862 flujo de múltiples productos de costo mínimo, 864 ex.
  - árbol de expansión de costo mínimo, véase corte mínimo de árbol de expansión mínimo, 721, 731 ej. grado mínimo, de un árbol B, 489 ciclo de peso medio mínimo, 680 pr. nodo mínimo, de un montón de Fibonacci, 508 cobertura de ruta mínima, 761 pr. árbol de expansión mínimo, 624–642 en algoritmo de aproximación para problema del viajante de comercio, 1112 Algoritmo de Borúuvka para, 641 sobre gráficas dinámicas, 637 ej. método genérico para, 625–630 Algoritmo de Kruskal para, 631–633 Algoritmo de Prim para, 634–636 relación con matroides, 437, 439–440 segundo mejor, 638 pr. árbol de expansión de peso mínimo, véase cubierta de vértice de peso mínimo de árbol de expansión mínimo, 1124–1127, 1139 menor de una matriz, 1224 cola de prioridad mínima, 162 en la construcción de códigos de Huffman, 431 en el algoritmo de Dijkstra, 661 en el algoritmo de Prim, 634, 636 falla, 449 pr. niño desaparecido, 1178 mod, 54, 928 operación de modificación, 230 aritmética modular, 54, 923 pr., 939–946 ej.
- equivalencia modular, 54, 1165 ex.
- exponenciación modular, 956
- EXPONENCIACIÓN MODULAR, 957**
- ecuaciones lineales modulares, 946–950
- SOLUCIÓN DE ECUACIONES LINEALES MODULARES, 949**
- módulo, 54, matriz Monge 928, 110 pr.
- secuencia monótona, 168
- monótonamente decreciente, 53
- monótonamente creciente, 53
- problema de Monty Hall, 1195 ej.
- heurística de movimiento al frente, 476 pr., 478 MST-KRUSKAL, 631
- MST-PRIM, 634
- MST-REDUCE, 639 pr.
- mucho mayor que (), 574 mucho menor que (), 783 flujo de múltiples productos básicos, 862–863 costo mínimo, 864 ex.
- computadora multinúcleo, 772 transformada rápida multidimensional de Fourier, 921 pr. multigraph, 1172 conversión a gráfico no dirigido equivalente, 593 ex.
- múltiplo, 927
  - de un elemento módulo n, 946–950
  - menos común, 939 ej.
  - escalar, 1220
- asignación múltiple, 21
- fuentes y sumideros múltiples, 712
- multiplicación de
  - números complejos, 83 ej.
  - método divide y vencerás para, 920 pr. de matrices, véase multiplicación de matrices de una cadena de matrices, 370–378 matriz-vector, multiproceso, 785–787, 789–790, 792 ej.
- módulo n (n), 940 de
  - polinomios, 899 método
- de multiplicación, 263–264 grupo
- multiplicativo módulo n, 941 inverso
- multiplicativo, módulo n, 949 instrucción de multiplicación, 23 MULTIPOP, 453 multiprocesador, 772 MULTIPUSH, 456

- serie múltiple, 1158  
 n. algoritmo multihilo, 10, 772–812  
     para calcular números de Fibonacci, 774–780 para  
     transformada rápida de Fourier, 804 ej.  
     Algoritmo de Floyd-Warshall, 797 ej. para  
     descomposición LU, 806 pr. para  
     descomposición LUP, 806 pr. para  
     inversión de matriz, 806 pr. para  
     la multiplicación de matrices, 792–797, 806 pr. para  
     transposición de matriz, 792 ej., 797 ej. para  
     producto matriz-vector, 785–787,  
     789–790, 792 ej.  
     para mediana, 805 ej.  
     para clasificación de combinación, 797–  
     805, 812 para combinación,  
     798–801 para estadísticas de  
     pedidos, 805 ej. para  
     tabiques, 804 ex. para el cálculo de  
     prefijos, 807 pr. para  
     clasificación rápida, 811  
     pr. para reducción, 807 pr. para un cálculo de  
     plantilla simple, 809 pr. para resolver sistemas  
     de ecuaciones lineales, 806 pr.  
     Algoritmo de Strassen, 795–796  
     composición de subprocessos múltiples, 784  
     fig. computación multiproceso, 777  
     programación multiproceso, 781–783  
     eventos mutuamente excluyentes, 1190  
     eventos mutuamente independientes, 1193
- N (conjunto de números naturales), 1158  
 algoritmo ingenuo, para coincidencia de cadenas, 988–  
 990 NAIVE-STRING-MATCHER, 988  
 spline cúbico natural, 840 pr.  
 números naturales (N), 1158  
     claves interpretadas como,  
 263 negativo de una matriz,  
 1220 ciclo de peso negativo  
     y restricciones de diferencia, 667 y  
     relajación, 677 ej. y caminos  
     más cortos, 645, 653–654, 692 ej., 700 ej.
- bordes de peso negativo, 645–646  
 vecino, 1172  
 vecindario, 735 ex. lista de  
 vecinos, 750  
 paralelismo anidado, 776, 805 pr.  
 cajas nido, 678 pr.
- flujo neto a través de un corte,  
 720 red  
     admissible, 749–750 flujo,  
     ver flujo red residual, 715–  
     719 para clasificación,  
     811  
 AL LADO DE LA PARTE SUPERIOR, 1031  
 NIL, 21  
 nodos, 1176  
     ver también vértice  
 variable no básica, 855  
 algoritmo multiproceso no determinista, 787 tiempo  
 polinomial no determinista, 1064 n. ver también NP
- gráfico no hamiltoniano, 1061 no  
 instancia, 1056 n. matriz  
 no invertible, 1223 restricción de  
 no negatividad, 851, 853 patrón de  
 cuerdas no superpuestas, 1002 ej. empuje no  
 saturado, 739, 745 matriz no  
 singular, 1223 potencia no  
 trivial, 933 ex. raíz cuadrada  
 no trivial de 1, módulo n, 956 propiedad sin  
 trayectoria, 650, 672 ecuación  
 normal, 837 norma de un  
 vector, 1222 NO función (:),  
 1071 no es un miembro del  
 conjunto (62), 1158 no es  
 equivalente (6), 54 Puerta  
 NO, 1070 NP (clase  
     de complejidad), 1049, 1064, 1066 ej.,  
     1105  
 NPC (clase de complejidad), 1050, 1069  
 NP-completo, 1050, 1069  
 NP-completitud, 9–10, 1048–1105  
     del problema de la satisfacibilidad del  
     circuito, 1070–1077  
     del problema de la camarilla, 1086–1089, 1105  
     de determinar si una fórmula booleana es una  
     tautología, 1086 ej. del  
     problema de la satisfacibilidad de la fórmula,  
     1079–1081, 1105 del  
     problema de coloración de gráficos, 1103 pr. del  
     problema de satisfacibilidad de la mitad 3-CNF,  
     1101 ej.  
     del problema del ciclo hamiltoniano, 1091–  
     1096, 1105 del  
     problema del camino hamiltoniano, 1101 ej.

- del problema de los conjuntos independientes, 1101
- pr. de programación lineal entera, 1101 ej. del problema del ciclo simple más largo, 1101 ej.
- prueba, de una lengua, 1078–1079 de programación con beneficios y plazos, 1104 pr. del problema
- de la cobertura del conjunto, 1122 ej. del problema de la partición de conjuntos, 1101 ej. del problema del subgrafo-isomorfismo, 1100 ej.
- del problema de suma de subconjuntos, 1097–1100 del problema de satisfacibilidad de 3-CNF, 1082–1085, 1105
- del problema del viajante de comercio, 1096–1097
- del problema de la cobertura de vértices, 1089–1091, 1105
- de programación entera 0-1, 1100 ej.
- NP-hard, 1069 n-
- set, 1161 n-
- tupla, 1162
- evento nulo, 1190
- árbol nulo, 1178
- vector nulo, 1224
- tamiz de campo numérico, 984 estabilidad numérica, 813, 815, 842 n-
- vector, 1218
- notación o, 50–51, 64
- notación O, 45 fig., 47–48, 64 notación O0 , 62 pr.  
Notación O, 62 pr.
- objeto, 21
  - asignación y liberación de, 243–244
  - implementación de matriz de, 241–246 paso como parámetro, 21 función objetivo, 664, 847, 851 valor objetivo, 847, 851 algoritmo de intercambio
  - de comparación ajeno, 208 pr. ocurrencia de un patrón, 985 OFF-LINE-MINIMUM, 583 pr.
  - almacenamiento en caché de problemas fuera de línea, 449 pr.
  - ancestros menos comunes, 584 pr.
  - mínimo, 582 pr.
- Notación omega, 45 fig., 48–49, 64 Algoritmo de aproximación 1, 1107
- método de un paso, 585
- correspondencia uno a uno, 1167 función uno a uno, 1167 problema de casco convexo en línea, 1039 ej. problema de contratación en línea, 139–141 ON-LINE-MAXIMUM, 140 programador de subprocessos múltiples en línea, 781 ON-SEGMENT, 1018 en función, 1167 tabla hash
- de direcciones abiertas, 269–277
  - con doble hashing, 272–274, 277 ex. con sondeo lineal, 272 con sondeo cuadrático, 272, 283 pr.
- intervalo abierto, 348
- OpenMP, 774
- árbol de búsqueda binario óptimo, 397–404, 413 OPTIMAL-BST, 402 valor objetivo óptimo, 851 solución óptima, 851 subconjunto óptimo, de una matriz, 439 subestructura óptima de selección de actividad, 416 de árboles binarios de búsqueda, 399–400 en programación dinámica, 379–384 del problema fraccionario de la mochila, 426 en algoritmos codiciosos, 425 de códigos Huffman, 435 de subsecuencias comunes más largas, 392–393 de multiplicación de cadenas de matrices, 373 de corte de varillas, 362 de caminos más cortos, 644–645, 687, 693–694 de caminos más cortos no ponderados, 382 de matroides ponderadas, 442 del problema de la mochila 0-1, 426 cobertura óptima de vértices, 1108 problema de optimización, 359, 1050, 1054 algoritmos de aproximación para, 10, 1106–1140
- y problemas de decisión, 1051
- Función OR ( \_ ), orden 697, 1071
- de un grupo, 945
- lineal, 1165
- parcial, 1165
- total, 1165
- par ordenado, 1161
- árbol ordenado, 1177
- orden de crecimiento, 28

- estadísticas de pedidos, 213–  
227 dinámico, 339–  
345 algoritmo multiproceso para, 805 ex.
- árbol de estadísticas de pedidos,  
339–345 consultas, 347 ej.
- Puerta OR, 1070
- origen, 1015
- o, en pseudocódigo, 22
- ortonormal, 842 OS-
- KEY-RANK, 344 ex.
- OS-RANK, 342 OS-
- SELECT, 341 fuera  
de grado, 1169
- producto exterior, 1222
- salida
  - de un algoritmo, 5 de
  - un circuito combinacional, 1071 de una
  - puerta lógica, 1070
- sistema sobredeterminado de ecuaciones lineales, 814
- desbordamiento
  - de una cola, 235
  - de una pila, 233
- vértice desbordante, 736
  - descarga de, 751
- intervalos superpuestos, 348
  - encontrando todo, 354
    - ej. punto de superposición máxima, 354 pr.
  - rectángulos superpuestos, 354 ej.
- subproblemas superpuestos, 384–386
- lema de sufijos superpuestos, 987
- P (clase de complejidad), 1049, 1055, 1059,  
1061 ej., 1105
- envoltura de paquete, 1037, 1047
- página en un disco, 486, 499 ej., 502 pr.
- par, ordenado, 1161
- conjuntos disjuntos por pares,  
1161 independencia por pares,  
1193 pares relativamente primos,  
931 palíndromo, 405 pr.
- Método de Pan para la multiplicación de matrices, 82 ej.
- algoritmo paralelo, 10, 772
  - véase también algoritmo multiproceso
- computadora paralela, 772
  - ideal, 779
- paralelo para, en pseudocódigo, 785–786
- paralelismo
  - lógico, 777
- de un cálculo multihilo, 780 anidados, 776
- de un
- algoritmo multihilo aleatorizado, 811 pr. bucle  
paralelo,
- 785–787, 805 pr. problema de
- programación de máquinas paralelas, 1136 pr. prefijo
- paralelo, 807 pr. máquina
- paralela de acceso aleatorio, 811 holgura
- paralela, 781 regla empírica,  
783
- paralelo, los hilos están lógicamente en, 778
- parámetro, 21
  - costos de paso, 107 pr.
- padre
  - en un árbol primero en anchura,  
594 en un cálculo de subprocesos múltiples,  
776 en un árbol enraizado,  
1176 PADRE,
  - 152 estructura de paréntesis de búsqueda primero en  
profundidad, 606 teorema de
  - paréntesis, 606 paréntesis de un producto matriz-cadena,  
370 árbol de
  - análisis, 1082 conjunto
  - parcialmente pedido,  
1165 pedido parcial,  
1165 PARTICIÓN<sup>8671pr.</sup>PARTICIÓN0
  - función de partición, 361 n.
  - partición, 171–173
    - alrededor de la mediana de 3 elementos, 185 ex.
    - Método de Hoare para, 185 pr.
    - algoritmo multiproceso para, 804 ex.
    - aleatorio, 179
  - partición de un conjunto, 1161,  
1164 Triángulo de Pascal, 1188
  - ex. ruta,
    - 1170 aumento, 719–720, 763 pr.
    - crítico, 657
    - hallazgo,
    - 569 hamiltoniano, 1066
    - ex. más largo, 382,
    - 1048 más corto, ver rutas más
    - cortas simples,  
1170 peso de,  
643 PATH, 1051, 1058
    - compresión de ruta, 569
    - cobertura de ruta, 761
    - pr. longitud de camino, de un árbol, 304 pr.,  
1180 ex. propiedad de relajación de ruta, 650, 673

- patrón, en combinación de cadenas, 985  
     no superpuestos, 1002 ex.  
 coincidencia de patrones, ver pena de coincidencia  
     de cadenas,  
         444 hashing perfecto, 277–282, 285  
     aceleración lineal perfecta, 780  
     coincidencia perfecta, 735 ej.  
     permutación, 1167  
         inversión de bits, 472 pr., 918  
         Josefo, 355 pr. k-  
         permutación, 126, 1184 lineal,  
             1229 pr. en su  
         lugar, 126  
         aleatorio, 124–128 de  
         un conjunto, 1184  
         aleatorio uniforme, 116, 125 matriz  
     de permutación, 1220, 1222 ej., 1226 ej.  
         LUP descomposición de, 827 ex.  
 PERMUTE-BY-CYCLIC, 129 ej.  
 PERMUTAR POR CLASIFICACIÓN, 125 PERMUTAR CON TODO, 129 ex.  
 PERMUTE-SIN-IDENTIDAD, 128 ej. estructura de  
     datos persistente, 331 pr., 482 PERSISTENT-TREE-INSERT, 331 pr.  
     Diagrama PERT, 657, 657 ej.  
 P-FIB, 776  
     fase, del algoritmo rebel-to-front, 758 función phi (.n/), 943  
 PISANO-DELETE, 526 pr. pivote  
     en programación lineal, 867, 869–870,  
         878 ej.  
     en descomposición LU, 821 en  
         clasificación rápida,  
 171 PIVOT,  
     869 plato, 485  
 P-MATRIX-MULTIPLY-RECURSIVE, 794 P-MERGE, 800 P-MERGE-SORT, 803 puntero, 21  
     implementación  
         de matriz de, 241–246 seguimiento, 295  
         punto  
         -representación de valor, 901 ángulo  
         polar, 1020 ej.  
     Heurística rho de Pollard, 976–980, 980 ej., 984 POLLARD-RHO, 976 polígono, 1020  
     ej. núcleo de, 1038 ej.  
     en forma de estrella, 1038  
     ej. acotado polilogarítmicamente, 57  
     polinomio, 55, 898 suma  
         de, 898  
         comportamiento asintótico de, 61 pr.  
         coeficiente representación de, 900 derivados  
         de, 922 pr. evaluación de,  
             41 pr., 900, 905 ej., 923 pr. interpolación por, 901, 906  
         ex. multiplicación de, 899, 903–905,  
             920 pr. representación de valor puntual de, 901  
     condición de crecimiento polinomial, 113  
     acotado polinomialmente, 55  
     relacionado polinomialmente, 1056  
     aceptación de tiempo polinomial, 1058  
     algoritmo de tiempo polinomial, 927, 1048 esquema  
     de aproximación de tiempo polinomial, 1107 para camilla  
         máxima, 1134 pr. computabilidad en  
         tiempo polinómico, 1056 decisión en tiempo  
         polinómico, 1059 reducibilidad en tiempo  
         polinómico (P), 1067, 1077 ej. Solvabilidad en tiempo  
             polinomial,  
                 1055 Verificación en tiempo polinomial, 1061–1066 POP, 233, 452 pop de una pila en tiempo de  
             ejecución, 188 pr.  
         árbol posicional, 1178 matriz definida positiva,  
         1225 problema de ubicación  
         de la oficina de correos, 225 pr.  
         caminata de árbol de orden posterior, 287  
         función potencial, 459  
         para límites inferiores, 478  
         método potencial, 459–463  
             para contadores binarios, 461–462  
             para estructuras de datos de conjuntos disjuntos, 575–581, 582 ej.  
             para tablas dinámicas, 466–471 para  
             montones de Fibonacci, 509–512, 517–518,  
                 520–522  
             para el algoritmo genérico push-relabel, 746 para min-heaps, 462 ex. para la  
             reestructuración de árboles rojo-negros, 474 pr. para  
             listas autoorganizadas con mover al frente,  
                 476 pr.  
             para operaciones de pila, 460–461  
         potencial, de una estructura de datos, 459  
         potencia  
             de un elemento, módulo n, 954–958

- kth, 933 ej. no
- trivial, 933 ej. serie de
- potencia, 108 pr. conjunto
- de potencia, 1161 Pr
- fg (distribución de probabilidad), 1190 PRAM, 811
- predecesor en
- árboles de
  - búsqueda binarios, 291–292 en un vector
  - de bits con un árbol binario superpuesto, 534 en un vector de bits con
  - un árbol superpuesto de altura constante, 535 de ancho
    - primeros árboles, 594 en B-
  - árboles, 497 ej. en listas enlazadas, 236 en árboles estadísticos de orden, 347 ej. en estructuras proto van Emde Boas, 544 ej. en árboles rojo-negros, 311 en árboles de caminos más cortos, 647 en árboles de Van Emde Boas, 551–552
  - PREDECESOR, 230 matriz predecesora, 685
  - subgrafo predecesor en
  - todos los pares de caminos más cortos, 685 en búsqueda en anchura, 600 en profundidad- primera búsqueda, 603 en rutas más cortas de fuente única, 647 propiedad de subgrafo predecesor, 650, 676 prioridad, 447 pr.
  - prefijo de una secuencia, 392 de una cadena ( ), 986
  - código de prefijo, 429
  - cálculo
    - de prefijo, 807 pr. función
    - de prefijo, 1003–1004 lema
  - de iteración de función
  - de prefijo, 1007 prefijo, 736, 765
  - preimagen de una matriz, 1228 pr.
  - preorder, total, 1165 preorder tree walk, 287
  - preclasificación, 1043
  - Algoritmo de Prim, 634–636, 642
  - con una matriz de adyacencia, 637 ej. en algoritmo de aproximación para
    - problema del vendedor ambulante, 1112
  - implementado con un montón de Fibonacci, 636
  - implementado con un montón mínimo, 636 con pesos de borde enteros, 637 ej.
- similitud con el algoritmo de Dijkstra, 634, 662 para gráficos dispersos, 638 pr. pruebas de primalidad, 965–975, 983
  - prueba de Miller-Rabin, 968–975, 983 prueba de pseudoprimalidad, 966–968 programa lineal primario, 880 agrupamiento primario, 272 memoria primaria, 484 función de distribución prima, 965 número primo, 928
- densidad de, 965–966
- teorema de los números primos, 965
- raíz primitiva de  $Z_{\text{norte}}$ , 955
- principal de la unidad, 907 principio de inclusión y exclusión, 1163 ej.
- PRINT-TODOS-PARES-RUTA MÁS CORTA, 685 PRINT-CUT-ROD-SOLUTION, 369 PRINT-INTERSECTING-SEGMENTS, 1028 ej.
- PRINT-LCS, 395 PRINT-OPTIMAL-PARENTS, 377 PRINT-PATH, 601
- PRINT-SET, 572 ej. cola de prioridad, 162–166 en la construcción de códigos de Huffman, 431 en el algoritmo de Dijkstra, 661 implementación de montón de, 162–166 límites inferiores para, 531 cola de prioridad máxima, 162 cola de prioridad mínima, 162, 165 ej. con extracciones monótonas, 168 en el algoritmo de Prim, 634, 636 implementación de la estructura proto van Emde Boas de, 538–545 implementación del árbol van Emde Boas de, 531–560 véase también árbol de búsqueda binaria, montón binomial, montón de fibonacci prueba comprobable probabilísticamente, 1105, 1140 análisis probabilístico, 115–116, 130–142 del algoritmo de aproximación para
  - Satisfacción de MAX-3-CNF, 1124 y entradas promedio, 28 de profundidad de nodo promedio en un árbol de búsqueda binario construido aleatoriamente, 304 pr. of balls and bins, 133–134 of birthday paradox, 130–133 of bucket sort, 201–204, 204 ej. de colisiones, 261 ej., 282 ej.

- de casco convexo sobre una distribución
- de casco ralo, 1046 pr.
- de comparación de archivos, 995 ej. de clasificación difusa de intervalos, 189 pr. de hashing con encadenamiento, 258–260 de altura de un árbol de búsqueda binario construido aleatoriamente, 299–303 de problema de contratación, 120–121, 139–141 de inserción en un árbol de búsqueda binario con claves iguales, 303 pr. de la sonda más larga con destino a hashing, 282 pr. de límite inferior para clasificación, 205 pr. de la prueba de primalidad de Miller-Rabin, 971–975 y algoritmos de subprocessos múltiples, 811 pr. del problema de la contratación en línea, 139–141 del hash de dirección abierta, 274–276, 277 ex. de partición, 179 ej., 185 ej., 187–188 pr. de hashing perfecto, 279–282 de la heurística rho de Pollard, 977–980 de conteo probabilístico, 143 pr. de clasificación rápida, 181–184, 187–188 pr., 303 ej. del algoritmo Rabin-Karp, 994 y algoritmos aleatorios, 123–124 de selección aleatoria, 217–219, 226 pr. de buscar una lista compacta, 250 pr. de tamaño de ranura encuadrado para encadenar, 283 pr. de clasificación de puntos por distancia desde el origen, 204 ej. de rayas, 135–139 de hashing universal, 265–268 conteo probabilístico, 143 pr. probabilidad, 1189–1196 función de densidad de probabilidad, 1196 distribución de probabilidad, 1190 función de distribución de probabilidad, 204 ej. secuencia de sondeo, 270 sondeo, 270, 282 pr. véase también sondeo lineal, sondeo cuadrático, problema de hash doble abstracto, 1054 computacional, 5–6 concreto, 1055 decisión, 1051, 1054 intratable, 1048 optimización, 359, 1050, 1054 solución a, 6, 1054–1055 tratable, 1048 procedimiento, 6, 16–17 producto . Q/, 1148 cartesiano, 1162 cruz, 1016 interior, 1222 de matrices, 1221, 1226 ex. externo, 1222 de polinomios, 899 regla de, 1184 flujo escalar, 714 ex. luchador profesional, 602 ej. contador de programa, 1073 programación, véase programación dinámica, programación lineal ancestro propio, 1176 descendiente propio, 1176 subgrupo propio, 944 subconjunto propio (), 1159 proto estructura de van Emde Boas, 538–545 grupo en, 538 comparado con árboles de van Emde Boas, 547 eliminación de, 544 inserción en, 544 máximo en, 544 ex. membresía en, 540–541 mínimo en, 541–542 predecesor en, 544 ex. sucesor en, 543–544 resumen en, 540 PROTO-VEB-INSERT, 544 PROTO-VEB-MIEMBRO, 541 PROTO-VEB-MINIMUM, 542 estructura proto-vEB, ver proto van Emde Boas estructura PROTO-VEB-SUCCESSOR, 543 método de poda y búsqueda, 1030 poda de un montón de Fibonacci, 529 pr. P-SCAN-1, 808 pr. P-SCAN-2, 808 pr. P-SCAN-3, 809 pr. P-SCAN-DOWN, 809 pr. P-SCAN-UP, 809 pr. pseudocódigo, 16, 20–22 pseudoinverso, 837 pseudoprímo, 966–968 PSEUDOPRIME, 967 generador de números pseudoaleatorios, 117 P-SQUARE-MATRIX-MULTIPLY, 793

- P-TRANSPOSE, 792 ex.
- clave pública, 959, 962
- criptosistema de clave pública, 958–965, 983
- PUSH
- operación de reetiquetado de inserción,
  - 739 operación de pila, 233, 452
  - inserción en una pila de tiempo de ejecución, 188
  - pr. operación de inserción (en algoritmos de reetiquetado de inserción), 738–739
  - no saturante, 739, 745
  - saturante, 739, 745
- algoritmo push-relabel, 736–760, 765 operaciones básicas en, 738–740 descargando un vértice desbordante de altura máxima, 760 ej. para encontrar una coincidencia bipartita máxima, 747 ej.
- brecha heurística para, 760 ej., 766
- algoritmo genérico, 740–748 con una cola de vértices desbordados, 759 ej. algoritmo de reetiquetado al frente, 748–760
- función cuadrática, 27
- sondeo cuadrático, 272, 283 pr. residuo cuadrático, 982 pr. cuantil, 223 ej. consulta, 230 cola, 232, 234–235 en búsqueda en amplitud, 595 implementada por pilas, 236 ej. implementación de lista enlazada de, 240 ej. prioridad, ver cola de prioridad en algoritmos push-relabel, 759 ej. clasificación rápida, 170–190 análisis de, 174–185 análisis de casos promedio de, 181–184 en comparación con clasificación por inserción, 178 ej. en comparación con radix sort, 199 con valores de elementos iguales, 186 pr. buena implementación en el peor de los casos de, 223 ex. “adversario asesino” para, 190 con método de mediana de 3, 188 pr. algoritmo multiproceso para, 811 pr. versión aleatoria de, 179–180, 187 pr. profundidad de pila de, 188 pr. versión recursiva de cola de, 188 pr. uso de ordenación por inserción en, 185 ej. análisis del peor caso de, 180–181
- QUICKSORT, 171
- QUICKSORT0 , 186 pr.
- cociente, 928
- R (conjunto de números reales), 1158
- Algoritmo de Rabin-Karp, 990–995, 1013 RABIN-KARP-MATCHER, 993 raza, 787–790
- RACE-EXAMPLE,
- 788 clasificación por radix, 197–200 en comparación con clasificación rápida, 199 RADIX-SORT , 198 árbol radix, 304 pr.
- RAM, 23–24
- RANDOM, 117
- máquina de acceso aleatorio, 23–24 paralelo, 811
- algoritmo aleatorio, 116–117, 122–130 y entradas promedio, 28 clasificación de comparación, 205 pr. para clasificación difusa de intervalos, 189 pr. para el problema de contratación, 123–124 para la inserción en un árbol de búsqueda binario con claves iguales, 303 pr. para la satisfacibilidad de MAX-3-CNF, 1123–1124, 1139
- Prueba de primalidad de Miller-Rabin, 968–975, 983
- multiproceso, 811 pr. para particiones, 179, 185 ej., 187–188 pr. para permutar una matriz, 124–128 Heurística rho de Pollard, 976–980, 980 ej., 984
- y análisis probabilístico, 123–124 quicksort, 179–180, 185 ex., 187–188 pr. redondeo aleatorio, 1139 para buscar una lista compacta, 250 pr. para selección, 215–220 hashing universal, 265–268 desempeño en el peor de los casos, 180 ex.
- ASISTENTE DE CONTRATACIÓN ALEATORIA, 124 PARTICIÓN ALEATORIA, 179 CLASIFICACIÓN RÁPIDA ALEATORIA, 179, 303 ej. relación con árboles de búsqueda binarios construidos aleatoriamente, 304 pr. redondeo aleatorio, 1139 SELECCIÓN ALEATORIA, 216 ALEATORIZAR EN EL LUGAR, 126

- árbol de búsqueda binario construido aleatoriamente,  
299–  
303, 304 pr. generador de números  
aleatorios, 117 permutación aleatoria,  
124–128 uniforme,  
116, 125 MUESTRA ALEATORIA,  
130 ex. muestreo aleatorio, 129 ej.,  
179 RANDOM-SEARCH, 143 pr.  
variable aleatoria, 1196–1201  
    indicador, véase indicador rango variable aleatoria,  
1167 de una  
    matriz, 1228 pr. rango  
        columna, 1223  
        completo,  
        1223 de una matriz, 1223, 1226  
        ex. de un nodo en un bosque disjunto, 569, 575, 581  
            ej.  
        de un número en un conjunto ordenado, 300, 339  
        en árboles estadísticos de orden, 341–343, 344–345 ej.  
        fila, 1223  
tasa de crecimiento,  
28 rayos, 1021 ex.  
RB-BORRAR, 324 RB-  
BORRAR-ARREGLAR, 326 RB-  
ENUMERAR, 348 ej.  
RB-INSERT, 315 RB-  
INSERT-FIXUP, 316 RB-JOIN,  
332 pr.  
RB-TRANSPLANT, 323  
alcanzabilidad en un gráfico ( ), 1170  
números reales (R), 1158  
reconstruir una solución óptima, en programación dinámica,  
    387 registro, 147  
rectángulo,  
354 ej. recurrencia,  
34, 65–67, 83–113 solución por el  
    método de Akra-Bazzi, 112–113 solución por el  
    método maestro, 93–97 solución por el  
    método del árbol de recurrencia, 88–93 solución  
    por el método de sustitución, 83–88  
ecuación de recurrencia, véase recursión de  
recurrencia, 30  
árbol de recurrencia, 37, 88–93  
    en prueba del teorema maestro, 98–100 y el  
    método de sustitución, 91–92  
SELECTOR-ACTIVIDAD-RECURSIVO, 419 caso  
recursivo, 65  
RECUSIVO-FFT, 911  
CADENA DE MATRIZ RECUSIVA, 385  
árbol rojo-negro, 308–338  
    aumento de, 346–347 en  
    comparación con árboles B, 484, 490  
    eliminación de, 323–330 para  
    determinar si algún segmento de línea se cruza,  
        1024 para  
    enumerar claves en un rango, 348 ej. altura de,  
        309 inserción en,  
        315–323 unión de, 332 pr.  
    clave máxima de, 311  
    clave mínima de, 311  
    predecesor en, 311  
    propiedades de, 308–  
        312 relajado, 311 ex.  
    reestructuración,  
        474 pr. rotación en, 312–  
        314 búsqueda en, 311  
    sucesor en, 311  
    véase también árbol  
    de intervalos, árbol de orden estadístico REDUCE,  
        807 pr. furgoneta de  
espacio reducido Emde Boas tree, 557 pr.  
reducibilidad, 1067–1068  
algoritmo de reducción, 1052, 1067  
función de reducción, 1067  
reducción, de una matriz, 807 pr.  
relación reflexiva, 1163  
reflexividad de notación asintótica, 51 región,  
factible, 847 condición de  
regularidad, 95 rechazo por  
un  
    algoritmo, 1058 por un  
    autómata finito, 996 RELABEL,  
740 vértice  
reetiquetado, 740  
operación de reetiquetado, en algoritmos push-relabel,  
    740, 745  
RELABEL-TO-FRONT, 755  
algoritmo rebel-to-front, 748–760 fase de,  
    758 relación,  
1163–1166 relativamente  
primo, 931 RELAX, 649  
relajación  
    de un borde, 648–650  
programación lineal, 1125

- montón relajado, 530  
 árbol rojo-negro relajado, 311 ex. tiempo  
 de lanzamiento, 447 pr.  
 resto, 54, 928 instrucción  
 de resto, 23 elevaciones repetidas  
 al cuadrado  
     para caminos más cortos de todos los pares,  
     689–691 para elevar un número a una potencia,  
 956 repetir, en pseudocódigo,  
 factor de repetición 20, de una cadena, 1012 pr.  
**REPETICIÓN-PARTIDO**, 1013 pr.  
 representante de un conjunto, 561  
**RESET**, 459 ex.  
 capacidad residual, 716, 719 borde  
 residual, 716 red  
 residual, 715–719 residuo, 54, 928,  
 982 pr. respetando un conjunto  
 de aristas, 626 arista de retorno, 779  
 retorno, en  
 pseudocódigo, 22 instrucción de  
 retorno, 23 reponderación  
 en rutas más  
     cortas de todos los pares, 700–702 en rutas  
     más cortas de fuente única, 679 pr.  
 heurística rho, 976–980, 980 ej., 984  
     .n-/algoritmo de aproximación, 1106, 1123 **RIGHT**, 152  
 right child, 1178  
 right-conversion, 314  
 ex. rayo horizontal derecho, 1021  
 ej.  
**ROTACIÓN A LA DERECHA**,  
 313 rotación derecha,  
 312 columna derecha,  
 333 pr. subárbol derecho,  
 1178 corte de varillas, 360–370, 390 ej.  
 raíz  
     de un árbol, 1176  
     de unidad, 906–907 de  
 Z <sub>norte</sub>, 955  
 árbol enraizado, 1176  
     representación de, 246–249 lista  
 raíz, de un montón de Fibonacci, 509 rotación  
  
 cíclico, 1012 ej. en  
     un árbol rojo-negro, 312–314  
 barrido rotacional, 1030–1038  
 redondeo, 1126  
     aleatorizado, 1139  
     orden de fila principal, rango  
     de 394 filas, vector  
     de 1223 filas,  
     criptosistema de clave pública 1218 RSA, 958–965, árbol  
     RS-vEB 983, 557 pr. regla  
     del producto, 1184 regla de  
     la suma, 1183 tiempo  
     de ejecución, 25 caso  
         promedio, 28, 116 mejor  
         caso, 29 ej., 49 esperado,  
         28, 117 de un algoritmo  
         gráfico, 588 y computación  
         multiproceso, 779–780 orden de crecimiento, 28 tasa  
         de crecimiento, 28 peor  
         de los casos, 27, 49  
  
 sabermetría, 412 n. borde  
 seguro, 626  
**MISMO COMPONENTE**, 563  
 espacio de muestra, 1189  
 muestreo, 129 ej., 179 SAT,  
 1079 datos  
 satelitales, 147, 229  
 satisfacibilidad, 1072, 1079–1081, 1105, 1123–  
 1124, 1127 ej., 1139 fórmula  
 satisfactoria , 1049, 1079 asignación  
 satisfactoria, 1072, 1079 borde saturado, 739  
 empuje de saturación,  
 739, 745 producto de flujo escalar,  
 714 ex. múltiplo escalar, escala  
 1220 en flujo máximo, 762  
 pr., 765  
     en rutas más cortas de fuente única, 679  
     pr. escanear, 807 pr.  
  
**ESCANEAR**, 807  
 pr. árbol de chivo  
 expiatorio, horario 338, 444,  
     1136 pr. event-point,  
 1023 planificador, para cálculos multiproceso, 777, 781–  
 783, 812 centralizado,  
 782 codicioso, 782  
     algoritmo de  
     robo de trabajo para, 812 programación,  
 443–446, 447 pr., 450, 1104 pr., 1136 pr.  
     Complemento de Schur, 820, 834

- Lema del complemento de Schur, 834  
 SCRAMBLE-SEARCH, 143 pr. talla de costura, 409 pr., 413 SEARCH, 230 búsqueda, 22 ex. búsqueda binaria, 39 ej., 799–800 en árboles de búsqueda binaria, 289–291 en árboles B, 491–492 en tablas hash encadenadas, 258 en listas compactas, 250 pr. en tablas de direcciones directas, 254 para un intervalo exacto, 354 ex. en árboles de intervalo, 350–353 búsqueda lineal, 22 ej. en listas enlazadas, 237 en tablas hash de direcciones abiertas, 270–271 en estructuras proto van Emde Boas, 540–541 en árboles rojo-negro, 311 en una matriz sin clasificar, 143 pr. en árboles Van Emde Boas, 550  
 árbol de búsqueda, ver árbol de búsqueda equilibrado, árbol de búsqueda binaria, árbol B, árbol de búsqueda exponencial, árbol de intervalo, árbol de búsqueda binaria óptima, árbol de estadísticas de orden, árbol rojo-negro, árbol splay, árbol 2-3, 2-3- Agrupación secundaria de 4 árboles, tabla hash secundaria 272, árbol de búsqueda de almacenamiento secundario 278, pilas 484–504, 502 pr. segundo mejor árbol de expansión mínimo, 638 pr. clave secreta, 959, 962 segmento, ver segmento dirigido, segmento de línea SEGMENTOS- INTERSECCIÓN, 1018 SELECCIONAR, 220 selección, 213 de actividades, 415–422, 450 y tipos de comparación, 222 en tiempo lineal esperado, 215–220 multihilo, 805 ej. en árboles estadísticos de orden, 340–341 en tiempo lineal en el peor de los casos, 220–224 clasificación por selección, 29 ej. vértice selector, 1093 bucle automático, 1168 lista autoorganizada, 476 pr., 478 gráfico semiconectado, 621 ex. centinela, 31, 238–240, 309  
 secuencia (hi) bitónica, 682 pr. finito, 1166 infinito, 1166 inversión en, 41 pr., 122 ej., 345 ej. sonda, 270 consistencia secuencial, 779, 812 algoritmo en serie versus algoritmo paralelo, 772 serialización, de un algoritmo de subprocessos múltiples, 774, 776 series, 108 pr., 1146–1148 hebras que están lógicamente en, 778 conjunto (fg), 1158–1163 cardinalidad (jj), 1161 convexo, 714 ex. diferencia (), 1159 independiente, 1101 pr. intersección (l), 1159 miembro (2), 1158 no miembro (62), 1158 unión (l), 1159 problema de cobertura de conjunto, 1117–1122, 1139 ponderado, 1135 pr. problema de partición de conjuntos, 1101 ej. sombra de un punto, 1038 ej. memoria compartida, 772 clasificación de Shell, 42 desplazamiento, coincidencia de cadenas, 985 instrucción de desplazamiento, 24 operador de cortocircuito, 22 RUTA MÁS CORTA, 1050 rutas más cortas, 7, 643–707 todos los pares, 644, 684–707 Bellman-Ford algoritmo para, 651–655 con caminos bitónicos, 682 pr. y búsqueda en amplitud, 597–600, 644 propiedad de convergencia de, 650, 672–673 y restricciones de diferencia, 664–670 Algoritmo de Dijkstra para, 658–664 en un gráfico acíclico dirigido, 655–658 en gráficos -densos, 706 pr. estimación de, 648 Algoritmo de Floyd-Warshall para, 693–697, 700 ej., 706 algoritmo de escala de Gabow para, 679 pr. Algoritmo de Johnson para, 700–706 como un programa lineal, 859–860 y rutas más largas, 1048

- por multiplicación de matrices, 686–693, 706–707 y ciclos de peso negativo, 645, 653–654, 692 ej., 700 ej. con bordes de peso negativo, 645–646 propiedad sin trayectoria de, 650, 672 subestructura óptima de, 644–645, 687, 693–694
- propiedad de relajación de ruta de, 650, 673 propiedad de subgrafo predecesor de, 650, 676 variantes del problema, 644 y relajación, 648–650 mediante elevación repetida al cuadrado, 689–691 destino único, 644 par único, 381, 644 fuente única , 643–683 árbol de, 647–648, 673–676 triángulo desigualdad de, 650, 671 en un gráfico no ponderado, 381, 597 propiedad de límite superior de, 650, 671–672 en un gráfico ponderado, 643 hermano, 1176 lado de un polígono, 1020 ej. firma, 960 ciclo simple, 1170 gráfico simple, 1170 ruta simple, 1170
- más largo, 382, 1048 polígono simple, 1020 ex. cálculo de plantilla simple, 809 pr. hashing uniforme simple, 259 simplex, 848 SIMPLEX, 871 algoritmo simplex, 848, 864– 879, 896–897 rutas más cortas de un solo destino, 644 ruta más corta de un solo par, 381, 644 como un programa lineal, 859–860 de una sola fuente caminos más cortos, 643–683
- Algoritmo de Bellman-Ford para, 651–655 con caminos bitónicos, 682 pr. y restricciones de diferencia, 664–670 Algoritmo de Dijkstra para, 658–664 en un gráfico acíclico dirigido, 655–658 en gráficos -densos, 706 pr. Algoritmo de escala de Gabow para, 679 pr. como programa lineal, 863 ej. y caminos más largos, 1048 singleton, 1161 gráfico conectado individualmente, 612 ej.
- lista enlazada individualmente, 236 ver también lista enlazada matriz singular, 1223 descomposición de valores singulares, 842 vértice sumidero, 593 ej., 709, 712 tamaño de la entrada de un algoritmo, 25, 926–927, 1055–1057 de un árbol binomial, 527 pr. de un circuito combinatorial booleano, 1072 de una camarilla, 1086 de un conjunto, 1161 de un subárbol en un montón de Fibonacci, 524 de una cubierta de vértice, 1089, 1108 lista de omisión, 338 holgura, 855 forma holgada, 846, 854–857 singularidad de, 876 holgura complementario, 894 pr. paralelo, 781 variable de holgura, 855
- ranura de una tabla de acceso directo, 254 de una tabla hash, 256 SLOW-ALL-PARS-SHORTEST-PATHS, 689 análisis suavizado, 897 ?Sócrates, 790 solución a un problema abstracto, 1054 básico, 866 a un problema computacional, 6 a un problema concreto, 1055 factible, 665, 846, 851 inviable, 851 óptimo, 851 a un sistema de ecuaciones lineales, 814 lista ordenada ordenada, 236 ver también vinculado lista
- clasificación, 5, 16–20, 30–37, 147–212, 797–805 bubblesort, 40 pr. clasificación de cubetas, 200–204 clasificación de columnas, 208 pr. clasificación de comparación, 191 clasificación de conteo, 194–197 borrosa, 189 pr. heapsort, 151– 169 clasificación por inserción, 12, 16–20

- k-clasificación, 207 pr.
- lexicográfico, 304 pr. en
- tiempo lineal, 194–204, 206 pr. límites
- inferiores para, 191–194, 211, 531 merge sort, 12, 30–37, 797–805 por algoritmos de intercambio de comparación ajenos, 208 pr.
- en su lugar, 17, 148, 206 pr. de
- puntos por ángulo polar, 1020 ej. límite inferior probabilístico para, 205 pr. clasificación rápida, 170–190
- clasificación radix, 197–200 clasificación por selección, 29 ej.
- Clasificación de
- Shell, 42
- estable, 196 tabla de tiempos de ejecución, 149 topológica, 8, 612–615, 623 usando un árbol de búsqueda binaria, 299 ej. con artículos de longitud variable, 206 pr. Lema de clasificación 0-1, 208 pr.
- red de clasificación, 811 vértice de origen, 594, 644, 709,
- 712 ley de expansión, 780
- árbol de expansión, 439, 624 cuello de botella, 640 pr. máximo, 1137 pr.
- verificación de, 642 véase también lapso mínimo de árbol de expansión, de un cálculo multiproceso, 779 gráfico disperso, 589
- caminos más cortos para todos los pares, 700–705 y algoritmo de Prim, 638 pr.
- distribución de casco disperso, 1046 pr.
- spawn, en pseudocódigo, 776–777 spawn edge, 778 speedup, 780 de un
- algoritmo aleatorio de subprocessos múltiples, 811 pr.
- husillo, 485 columna
- vertebral de un autómata de emparejamiento de cuerdas, 997 fig. de
- un trato, 333 pr. árbol splay, 338, 482
- spline, 840
- pr. división de nodos de árbol B, 493–495 de 2-3-4 árboles, 503 pr. división de sumas, 1152–1154
- golpe espurio, 991
- matriz cuadrada, 1218
- MATRIZ-CUADRADA-MULTIPLICAR, 75, 689
- MATRIZ-CUADRADA-MULTIPLICAR-RECURSIVA, 77
- cuadrado de un grafo dirigido, 593 ej. raíz cuadrada, módulo a primo, 982 pr. cuadratura, repetido
- para caminos más cortos de todos los pares, 689–691 para elevar un número a una potencia, 956
- estabilidad numérica, 813, 815, 842 de algoritmos de clasificación, 196, 200
- ej. stack, 232–233
- en el escaneo de Graham, 1030 implementado por colas, 236 ej.
- implementación de lista enlazada de, 240 ej.
- operaciones analizadas por método contable, 457–458
- operaciones analizadas por análisis agregado, 452–454
- operaciones analizadas por el método potencial, 460–461 para
- la ejecución del procedimiento, 188 pr.
- sobre almacenamiento secundario, 502 pr.
- STACK-EMPTY, 233
- desviación estándar, 1200
- codificación estándar (hi), 1057 forma estándar, 846, 850–854 polígono en forma de estrella, 1038 ej. estado de inicio, 995 tiempo de inicio, 415
- estado de un autómata finito, 995 gráfico estático, 562 n. juego
- estático de llaves, 277 hilos
- estáticos, 773 esténcil, 809 pr. cálculo de plantilla, 809 pr.
- Aproximación de Stirling, 57 gestión de almacenamiento, 151, 243–244, 245 ej., 261 ej.
- almacenar instrucción, 23
- a horcadas, 1017 hilo, 777
- final, 779
- independiente, 789
- inicial, 779
- lógicamente en paralelo, 778

- lógicamente en serie, 778
- Algoritmo de Strassen, 79–83, 111–112
  - multiproceso, 795–796 rayas, 135–139 estrictamente
  - decreciente, 53 estrictamente
  - creciente, 53 cadena, 985, 1184 coincidencia de
  - cadena, 985–1013 basado en
    - factores de repetición, 1012 pr. por autómatas finitos, 995–1002 con caracteres de espacio, 989 ej., 1002 ej.
    - Algoritmo de Knuth-Morris-Pratt para, 1002–1013 algoritmo ingenuo para, 988–990
    - Algoritmo de Rabin-Karp para, 990–995, 1013
  - autómata de coincidencia de cadenas, 996–1002, 1002 ex.
  - componente fuertemente conectado, 1170 descomposición en, 615–621, 623
  - COMPONENTES FUERTEMENTE CONECTADOS, 617**
  - grafo fuertemente conectado, 1170
  - subgrafo, 1171
    - predecesores, ver subgrafo predecesores problema de isomorfismo de subgrafo, 1100 ej. subgrupo, 943–946 subrayecto, 1170 gráfico de subproblemas, 367–368 llamada de subrutina,
      - 21, 23, 25 n. ejecución, 25 n. subsecuencia,
  - 391 subconjunto (), 1159, 1161
    - familia hereditaria de, 437
    - familia independiente de, 437
  - SUBSET-SUM, 1097**
    - problema de suma de subconjuntos
      - algoritmo de aproximación para, 1128–1134, 1139
      - NP-completitud de, 1097–1100 con objetivo unario, 1101 ej.
      - método de sustitución, 83–88 y árboles de recurrencia, 91–92
      - subcadena, 1184
      - instrucción de resta, 23 resta de matrices, 1221 subárbol, 1176 mantenimiento de tamaños de, en orden-árboles estadísticos, 343–344
  - éxito, en un juicio de Bernoulli, 1201
  - sucesor
    - en árboles de búsqueda binarios, 291–292 en un vector de bits con un árbol binario
  - superpuesto, 533 en un vector de bits con un árbol superpuesto de altura constante, 535
  - encontrar *ith*, de un nodo en un árbol de estadística de orden, 344
    - ej. en listas enlazadas, 236 en árboles estadísticos de orden, 347 ej. en estructuras proto van Emde Boas, 543–544 en árboles rojo-negros, 311 en árboles Van Emde Boas, 550–551
  - SUCESOR, 230** tal que (W), 1159 sufijo ( ), 986 sufijo-función función, 996 sufijo-función desigualdad, 999 sufijo-función lema recursivo, 1000 suma . P/, 1145 cartesiano, 906 ej. infinito, 1145 de matrices, 1220 de polinomios, 898 regla de, 1183 telescopico, 1148 SUM-  
**ARRAYS, 805** pr.
  - SUMA-ARRAYS0, 805** pr.
  - resumen
    - en un vector de bits con un árbol superpuesto de altura constante, 534
    - en proto estructuras de van Emde Boas, 540 en árboles de van Emde Boas, 546
  - suma, 1145–1157 en notación
    - asintótica, 49–50, 1146 delimitación, 1149–1156 fórmulas y propiedades de, 1145–1149 linealidad de, 1146 lema de suma, 908
    - supercomputadora, 772
    - tiempo superpolinomio, 1048 supersink, 712 supersource, 712 sobreyección, 1167 SVD, 842
    - barrido, 1021–1029, 1045 pr.
    - rotacional, 1030–1038

- línea de barrido, 1022
- estado de la línea de barrido, 1023–1024
- tabla de símbolos, 253, 262, 265
- diferencia simétrica, 763 pr. matriz
- simétrica, 1220, 1222 ej., 1226 ej. matriz definida positiva
- simétrica, 832–835,  
842
- relación simétrica, 1163 simetría
- de notación „, 52 sincronización, en
- pseudocódigo, 776–777 sistema de
- restricciones de diferencia, 664–670 sistema de
- ecuaciones lineales, 806 pr., 813–827, 840 pr.
- TABLA-ELIMINAR**, 468
- INSERCIÓN DE MESA**, cola  
464
  - de una distribución binomial, 1208–1215 de una
  - lista enlazada, 236 de una
  - cola, 234 recursión
  - de cola, 188 pr., 419 COLA-
  - RECURSIVA-CLASIFICACIÓN RÁPIDA**, 188 pr.
  - objetivo, 1097
  - Algoritmo de ancestros menos comunes fuera de línea
    - de Tarjan, 584 pr. tarea,  
443
  - Biblioteca paralela de tareas, 774
  - programación de tareas, 443–446, 448 pr., 450
  - tautología, 1066 ej., 1086 ej.
  - Serie de Taylor, 306 pr.
  - serie telescopica, 1148 suma
  - telescopica, 1148 prueba de
  - primalidad, 965–975, 983 de
    - pseudoprimalidad, 966–968
  - texto, en coincidencia de cadenas, 985
  - luego cláusula, 20 n.
  - Notación theta, 44–47, 64
  - subprocesos,
  - 773 Bloques de creación de subprocesos,
  - 774 3-CNF, 1082
  - 3-CNF-SAT, 1082
  - Satisfacibilidad de 3-CNF, 1082–1085, 1105
    - algoritmo de aproximación para, 1123–1124,  
1139
    - y 2-CNF satisfacibilidad, 1049 3-COLOR,
  - 1103 pr. 3-forma normal
  - conjuntiva, 1082
- restricción estricta, 865
- tiempo, ver dominio de tiempo
- de tiempo de ejecución,
- 898 compensación de tiempo-memoria,
- 365 marca de tiempo, 603, 611 ej.
- Matriz Toeplitz, 921 pr. a, en
- pseudocódigo, 20 TOP, 1031
- método de
- arriba hacia abajo, para programación dinámica,  
365
- parte superior de una
- pila, 232 clasificación topológica, 8, 612–615,
  - 623 en el cálculo de las rutas más cortas de una sola fuente  
en un dag,
- 655 TOPOLOGICAL-SORT, 613
- orden total, 1165
- longitud de ruta total, 304 pr.
- pedido anticipado total,
- 1165 relación total, gira  
1165
  - bitónico, 405 pr.
  - Euler, 623 pr., 1048 de un  
gráfico, 1096
  - seguimiento,
  - 486 manejabilidad,
  - 1048 puntero de
  - seguimiento, 295 función de transición, 995, 1001–1002, 1012  
ej. clausura transitiva, 697–699 y
    - multiplicación de matrices booleanas, 832 ej. de gráficos  
dinámicos, 705 pr., 707
  - CIERRE TRANSITIVO**, 698 relación  
transitiva, 1163 transitividad de
  - notación asintótica, 51 TRASPLANTE, 296, 323
  - transposición conjugada, 832 ej.
  - de un grafo
    - dirigido, 592 ej. de una
    - matriz, 1217 de una matriz,  
multiproceso, 792 ej.
    - simetría de transposición de notación asintótica,
  - 52 algoritmo de aproximación del problema del viajante de  
comercio para, 1111–1117,  
1139
    - euclíadiana bitónica, 405 pr. cuello  
de botella, 1117 ej.
    - NP-completitud de, 1096–1097 con la  
desigualdad triangular, 1112–1115 sin la desigualdad  
triangular, 1115–1116

- recorrido de un árbol, 287, 293 ex., 342, 1114 treap, 333 pr., 338 TREAP-  
 INSERT, 333 pr. árbol, 1173–1180 árboles AA, 338 AVL, 333 pr., 337 binario, ver árbol binario binomial, 527 pr. bisección de, 1181 pr. anchura primero, 594, 600 árboles B, 484–504 decisión, 192–193 profundidad primero, 603 diámetro de, 602 ex. dinámico, 482 libre, 1172–1176 recorrido completo, 1114 fusión, 212, 483 montón, 151–169 altura equilibrada, 333 pr. altura de, intervalo 1177, 348–354 k-vecino, 338 extensión mínima, ver extensión mínima árbol búsqueda binaria óptima, 397–404, 413 estadística de orden, 339–345 análisis, 1082 recursividad, 37, 88–93 rojo-negro, ver árbol rojo-negro enraizado, 246–249, 1176 chivo expiatorio, 338 búsqueda, ver árbol de búsqueda caminos más cortos, 647–648, 673–676 expansión, consulte árbol de expansión mínimo, expansión de árbol de expansión, 338, 482 treap, 333 pr., 338 2-3, 337, 504 2-3-4, 489, 503 pr. van Emde Boas, 531–560 paseo de, 287, 293 ej., 342, 1114 árboles de peso equilibrado, 338 TREE-DELETE, 298, 299 ej., 323–324 borde de árbol, 601, 603, 609 TREE-INSERT, 294, 315 ÁRBOL-MÁXIMO, 291 ÁRBOL-MÍNIMO, 291
- TREE-PREDECESOR, 292 TREE-SEARCH, 290 TREE-SUCCESSOR, 292 tree walk, 287, 293 ex., 342, 1114 trial, Bernoulli, 1201 trial division, 966 Triangle desigualdad, 1112 para caminos más cortos, 650, 671 matriz triangular, 1219, 1222 ej., 1225 ej. tricotomía, intervalo, 348 propiedad de tricotomía de los números reales, 52 sistemas lineales triadiagonales, 840 pr. matriz tri diagonal, 1219 trie (árbol de base), 304 pr. y-rápido, 558 pr.
- TRIM, 1130 recortar una lista, 1130 divisor trivial, 928 asignación de verdad, 1072, 1079 tabla de verdad, 1070 TSP, 1096 tupla, 1162 factor de giro, 912 2-CNF-SAT, 1086 ex. 2-CNF satisfacibilidad, 1086 ej. y satisfacibilidad 3-CNF, 1049 método de dos pasos, 571 montón 2-3-4, 529 pr. 2-3-4 árbol, 489 unión, 503 pr. división, 503 pr. 2-3 árbol, 337, 504
- unario, 1056 programa lineal ilimitado, 851 instrucción de bifurcación incondicional, 23 conjunto incontable, 1161 sistema indeterminado de ecuaciones lineales, 814 desbordamiento de una cola, 234 de una pila, 233 gráfico no dirigido, 1168 punto de articulación de, 621 pr. componente biconectado de, 621 pr. puente de, 621 pr. camarilla en, 1086 coloración de, 1103 pr., 1180 pr.

- calcular un árbol de expansión mínimo en, 624–642
- convirtiendo a, de un multigrafo, 593 ex. d-regular, 736 ej. rejilla, 760 pr.
- hamiltoniano, 1061 conjunto
- independiente de, 1101 pr.
- coincidencia de, 732
- no hamiltoniano, 1061
- cubierta de vértice de, 1089, 1108 véase también gráfico versión no dirigida de un gráfico dirigido, 1172 hashing
- uniforme, 271 distribución de probabilidad uniforme, 1191–1192 permutación aleatoria uniforme, 116, 125 unión
  - de conjuntos dinámicos, véase
  - unión de lenguas, 1058
  - de conjuntos ( $\cup$ ), 1159
- UNION, 505, 562
  - implementación de bosque conjunto disjunto de, 571 implementación de lista enlazada de, 565–567,
  - 568 ex. unión por
- rango, 569 factorización única de enteros, 931 unidad (1), 928
  - unión de montones de Fibonacci, 511–512 de
  - montones, 506 de listas
  - enlazadas, 241 ej. de 2-3-4
  - montones, 529 pr. unidad de matriz triangular inferior, 1219
  - tarea de unidad de tiempo, 443 unidad de matriz triangular superior, 1219 vector unitario, 1218 colección universal de funciones hash, 265
  - hashing universal, 265–268 sumidero
    - universal, 593 ex. universo, 1160 de claves en árboles van Emde Boas, 337
  - tamaño del universo, 532 vértice no coincidente, 732
  - lista enlazada no ordenada, 236
    - véase también lista enlazada
  - hasta, en pseudocódigo, 20
  - rutas simples más largas no ponderadas, 382
  - rutas más cortas no ponderadas, 381
  - límite superior, 47
- propiedad de límite superior, 650, 671–672
- mediana superior, 213
- cuadrada superior  $p^n$ , matriz <sup>548</sup> raíz triangular superior, 1219, 1225 ex.
- turno válido, valor 985
  - de un flujo, 710
  - de una función, 1166
  - objetivo, 847, 851
- valor sobre el jugador de reemplazo, 411 pr.
- Matriz de Vandermonde, 902, 1226 pr.
- árbol van Emde Boas, 531–560
  - agrupar en, 546
  - en comparación con estructuras proto van Emde Boas, 547
  - eliminación de, 554–556
  - inserción en, 552–554
  - máximo en, 550
  - membresía en, 550
  - mínimo en, 550
  - predecesor en, 551–552 con
  - espacio reducido, 557 pr. sucesor en, 550–551 resumen en, 546 Var $\sigma$  (varianza), 1199 variable
    - básico, 855
    - entrando, 867
    - saliendo, 867
    - no básico, 855
    - en pseudocódigo, 21
    - aleatorio, 1196–1201
    - holgura, 855 véase también indicador de variable aleatoria
    - código de longitud variable, 429 varianza, 1199 de una distribución binomial, 1205 de una distribución geométrica, 1203
- VEB-ÁRBOL-137 VACÍO-INSERTAR, 553
- árbol vEB, ver árbol van Emde Boas
- VEB-ÁRBOL-ELIMINAR, 554
- VEB-ÁRBOL-INSERTO, 553
- VEB-ÁRBOL-MÁXIMO, 550
- VEB-ÁRBOL-MIEMBRO, 550
- VEB-ÁRBOL-MÍNIMO, 550
- VEB-ÁRBOL-ANTECESOR, 552
- VEB-ÁRBOL-SUCESOR, 551

- vector, 1218, 1222–1224
  - convolución de, 901
  - producto cruzado de, 1016
  - ortonormal, 842 en el
  - plano, 1015
- diagrama de Venn, 1160
- verificación, 1061–1066 de
  - árboles de expansión, 642
  - algoritmo de verificación, 1063
  - vértice
    - punto de articulación, 621 pr.
    - atributos de, 592
    - capacidad de, 714 ex.
    - en un gráfico, 1168
    - intermedio, 693
    - aislado, 1169
    - desbordante, 736 de
    - un polígono, 1020 ej.
    - reetiquetado,
    - selector 740, 1093
  - cubierta de vértice, 1089, 1108, 1124–1127, 1139
- VERTEX-COVER, 1090
- problema de cobertura de vértices
  - algoritmo de aproximación para, 1108–1111, 1139
  - NP-completitud de, 1089–1091, 1105 conjunto de vértices, 1168
  - violación, de una restricción de igualdad, 865
- memoria virtual, 24
- algoritmo de Viterbi, 408 pr.
- VORP, 411 pr.
- paseo de un árbol, 287, 293 ej., 342, 1114
- dualidad débil, 880–881, 886 ej., 895 pr. peso de un corte,
  - 1127 ex. de un borde,
  - 1191 medio, 680 pr.
  - de un camino,
  - 643 árbol de
- peso equilibrado, 338, 473 pr.
- emparejamiento bipartito ponderado, 530
- matroide ponderado, 439–442
- mediana ponderada, 225 pr.
- problema ponderado de cobertura de conjuntos, 1135 pr. heuristic de unión
- ponderada, 566 cobertura de vértice ponderado, 1124–1127, 1139
  - función de peso para un gráfico, 591
- en una matroide ponderada, 439
- while, en pseudocódigo, 20
- teorema del camino blanco, 608 vértice blanco, 594, 603
- widget, 1092
- cable, 1071
- TESTIGO, 969
- testigo, de la composición de un número, 968 ley de trabajo, 780 trabajo, de
- un cálculo multiproceso, 779 algoritmo de programación de robo de trabajo, 812 tiempo de ejecución en el peor de los casos, 27, 49
- Mejora de Yen al algoritmo Bellman-Ford, 678 pr.
  - prueba rápida y, 558
  - pr.
  - Cuadro joven, 167 pr.
- Z (conjunto de enteros), 1158
- Zn (clases de equivalencia módulo n), 928 Z
  - (elementos del grupo multiplicativo módulo n), 941 ZC (elementos distintos de cero de Zn), 967 matriz cero, 1218 cero de un
  - polinomio módulo a primo, 950 ex. 0-1 programación entera, 1100 ej., 1125 0-1 problema de mochila, 425, 427 ej., 1137 pr., 1139
- Lema de clasificación 0-1, 208
- pr. zonk, 1195 ex.