**Algoritmos**

**Jeff Erickson**

**Algoritmos**

**Jeff Erickson**

**Edición 0 (borrador pre-publicación)** — 30 de diciembre de 2018  
**Edición ½ (borrador pre-publicación)** — 9 de abril de 2019  
**1ª edición en rústica** — 13 de junio de 2019  
1 2 3 4 5 6 7 8 9 — 27 26 25 24 23 22 21 20 19

**ISBN:** 978-1-792-64483-2 (rústica)

© Copyright 2019 Jeff Erickson

Esta obra está disponible bajo una Licencia Creative Commons Atribución 4.0 Internacional.  
Para detalles de la licencia, ver http://creativecommons.org/licenses/by/4.0/.

Descarga este libro en http://jeffe.cs.illinois.edu/teaching/algorithms/  
o http://algorithms.wtf  
o https://archive.org/details/Algorithms-Jeff-Erickson

Por favor reporta errores en https://github.com/jeffgerickson/algorithms

*Partes de nuestra programación son reproducidas mecánicamente,  
y ahora comenzamos nuestro día de transmisión.*

**Para Kim, Kay y Hannah  
con amor y admiración**

**Y para Erin  
con agradecimiento  
por romper su promesa**

*Incipit prologus in libro alghoarismi de practica arismetrice.*  
— Ioannis Hispalensis [¿Juan de Sevilla?],  
*Liber algorismi de pratica arismetrice* (c.1135)

*¿Te diré, amigo mío, cómo llegarás a entenderlo?  
Ve y escribe un libro sobre ello.*  
— Henry Home, Lord Kames (1696–1782),  
en una carta a Sir Gilbert Elliot

*El individuo siempre se equivoca. Diseñó muchas cosas, e involucró a otras personas como coadyuvantes, discutió con algunos o todos, se equivocó mucho, y algo se hace; todos avanzan un poco, pero el individuo siempre se equivoca. Resulta algo nuevo y muy diferente a lo que se prometió a sí mismo.*  
— Ralph Waldo Emerson, "Experience", *Essays, Second Series* (1844)

*Lo que he esbozado arriba es el contenido de un libro cuya realización del plan básico y la incorporación de cuyos detalles quizás sería imposible; lo que he escrito es un segundo o tercer borrador de una versión preliminar de este libro*  
— Michael Spivak, prefacio de la primera edición de *Differential Geometry, Volume I* (1970)

**Prefacio**

**Acerca de Este Libro**

Este libro de texto surgió de una colección de notas de conferencia que escribí para varias clases de algoritmos en la Universidad de Illinois en Urbana-Champaign, que he estado enseñando aproximadamente una vez al año desde enero de 1999. Impulsado por cambios en nuestro currículo de teoría de pregrado, emprendí una revisión importante de mis notas en 2016; este libro consiste en un subconjunto de mis notas revisadas sobre el material del curso más fundamental, reflejando principalmente el contenido algorítmico de nuestro nuevo curso requerido de teoría a nivel de tercer año universitario.

**Prerrequisitos**

Las clases de algoritmos que enseño en Illinois tienen dos prerrequisitos significativos: un curso de matemáticas discretas y un curso de estructuras de datos fundamentales. En consecuencia, este libro de texto probablemente no es adecuado para la mayoría de los estudiantes como un primer curso en estructuras de datos y algoritmos. En particular, asumo al menos familiaridad pasajera con los siguientes temas específicos:

• **Matemáticas discretas:** Álgebra de secundaria, identidades logarítmicas, teoría ingenua de conjuntos, álgebra booleana, lógica de predicados de primer orden, conjuntos, funciones, equivalencias, órdenes parciales, aritmética modular, definiciones recursivas, árboles (como objetos abstractos, no estructuras de datos), grafos (vértices y aristas, no gráficas de funciones).

• **Técnicas de demostración:** directa, indirecta, contradicción, análisis exhaustivo de casos, e inducción (especialmente inducción "fuerte" y "estructural"). El Capítulo 0 usa inducción, y siempre que el Capítulo n−1 use inducción, también lo hace el Capítulo n.

• **Conceptos de programación iterativa:** variables, condicionales, bucles, registros, indirección (direcciones/punteros/referencias), subrutinas, recursión. No asumo fluidez en ningún lenguaje de programación particular, pero sí asumo experiencia con al menos un lenguaje que soporte tanto indirección como recursión.

• **Tipos de datos abstractos fundamentales:** escalares, secuencias, vectores, conjuntos, pilas, colas, mapas/diccionarios, mapas/diccionarios ordenados, colas de prioridad.

• **Estructuras de datos fundamentales:** arrays, listas enlazadas (simples y dobles, lineales y circulares), árboles de búsqueda binaria, al menos una forma de árbol de búsqueda binaria balanceado (como árboles AVL, árboles rojo-negro, treaps, skip lists, o splay trees), tablas hash, heaps binarios, y más importante, la diferencia entre esta lista y la anterior.

• **Problemas computacionales fundamentales:** aritmética elemental, ordenamiento, búsqueda, enumeración, recorrido de árboles (preorden, inorden, postorden, por niveles, etc.).

• **Algoritmos fundamentales:** algoritmia elemental, búsqueda secuencial, búsqueda binaria, ordenamiento (selección, inserción, merge, heap, quick, radix, etc.), búsqueda en anchura y profundidad en árboles (al menos binarios), y más importante, la diferencia entre esta lista y la anterior.

• **Análisis elemental de algoritmos:** Notación asintótica (o, O, Θ, Ω, ω), traducir bucles en sumas y llamadas recursivas en recurrencias, evaluar sumas y recurrencias simples.

• **Madurez matemática:** facilidad con abstracción, definiciones formales (especialmente recursivas), y demostraciones (especialmente inductivas); escribir y seguir argumentos matemáticos; reconocer y evitar disparates sintácticos, semánticos y/o lógicos.

El libro cubre brevemente parte de este material prerrequisito cuando surge en contexto, pero más como recordatorio que como buena introducción. Para un resumen más completo, recomiendo encarecidamente las siguientes referencias disponibles gratuitamente:

**Referencias Adicionales**

• Margaret M. Fleck. *Building Blocks for Theoretical Computer Science*. Versión 1.3 (enero 2013) o posterior disponible en http://mfleck.cs.illinois.edu/building-blocks/.

• Eric Lehman, F. Thomson Leighton, y Albert R. Meyer. *Mathematics for Computer Science*. Revisión de junio 2018 disponible en https://courses.csail.mit.edu/6.042/spring18/. (Recomiendo encarecidamente buscar la revisión más reciente.)

• Pat Morin. *Open Data Structures*. Edición 0.1Gβ (enero 2016) o posterior disponible en http://opendatastructures.org/.

• Don Sheehy. *A Course in Data Structures and Object-Oriented Design*. Revisión de febrero 2019 o posterior disponible en https://donsheehy.github.io/datastructures/.

**Referencias Adicionales**

Por favor no te restrinjas a esta o cualquier otra referencia única. Los autores y lectores aportan sus propias perspectivas a cualquier material intelectual; ningún instructor "conecta" con cada estudiante, o incluso con cada estudiante muy fuerte. Encontrar al autor que más efectivamente transmite su intuición a tu cabeza requiere algo de esfuerzo, pero ese esfuerzo se paga generosamente a largo plazo.

Las siguientes referencias han sido fuentes particularmente valiosas de intuición, ejemplos, ejercicios e inspiración; esto no pretende ser una lista completa.

• Alfred V. Aho, John E. Hopcroft, y Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. (Usé este libro de texto como estudiante de pregrado en Rice y de nuevo como estudiante de maestría en UC Irvine.)

• Boaz Barak. *Introduction to Theoretical Computer Science*. Borrador de libro de texto, revisado más recientemente en junio 2019. (No es el libro de texto de ciencias de la computación teóricas de tu abuelo, y tanto mejor por ello; el hecho de que sea gratuito es un bono delicioso.)

• Thomas Cormen, Charles Leiserson, Ron Rivest, y Cliff Stein. *Introduction to Algorithms*, tercera edición. MIT Press/McGraw-Hill, 2009. (Usé la primera edición como asistente de enseñanza en Berkeley.)

• Sanjoy Dasgupta, Christos H. Papadimitriou, y Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006. (Probablemente el más cercano en contenido a este libro, pero considerablemente menos verboso.)

• Jeff Edmonds. *How to Think about Algorithms*. Cambridge University Press, 2008.

• Michael R. Garey y David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

• Michael T. Goodrich y Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2002.

• Jon Kleinberg y Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005. Pídelo prestado de la biblioteca si puedes.

• Donald Knuth. *The Art of Computer Programming*, volúmenes 1–4A. Addison-Wesley, 1997 y 2011. (Mis padres me dieron los primeros tres volúmenes para Navidad cuando tenía 14 años. Lamentablemente, no los leí realmente hasta mucho después.)

• Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989. (Usé este libro de texto como asistente de enseñanza en Berkeley.)

• Ian Parberry. *Problems on Algorithms*. Prentice-Hall, 1995 (fuera de impresión). Descargable desde https://larc.unt.edu/ian/books/free/license.html después de que aceptes hacer una pequeña donación caritativa. Por favor honra tu acuerdo.

• Robert Sedgewick y Kevin Wayne. *Algorithms*. Addison-Wesley, 2011.

• Robert Endre Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

• Notas de clase de mis propias clases de algoritmos en Berkeley, especialmente las enseñadas por Dick Karp y Raimund Seidel.

• Notas de conferencia, diapositivas, tareas, exámenes, video conferencias, artículos de investigación, publicaciones de blog, preguntas y respuestas de StackExchange, podcasts, y MOOCs completos disponibles gratuitamente en la web por innumerables colegas alrededor del mundo.

**Acerca de los Ejercicios**

Cada capítulo termina con varios ejercicios, la mayoría de los cuales he usado al menos una vez en una tarea, sección de discusión/laboratorio, o examen. Los ejercicios no están ordenados por dificultad creciente, sino (generalmente) agrupados por técnicas o temas comunes. Algunos problemas están anotados con símbolos como sigue:

• ♥ **Los corazones rojos** indican problemas particularmente desafiantes; muchos de estos han aparecido en exámenes de calificación para estudiantes de doctorado en Illinois. Un pequeño número de problemas realmente difíciles están marcados con ♥ **corazones grandes**.

• ♦ **Los diamantes azules** indican problemas que requieren familiaridad con material de capítulos posteriores, pero temáticamente pertenecen donde están. Los problemas que requieren familiaridad con material anterior no están marcados, sin embargo; el libro, como la vida, es acumulativo.

• ♣ **Los tréboles verdes** indican problemas que requieren familiaridad con material fuera del alcance de este libro, como máquinas de estado finito, álgebra lineal, probabilidad, o grafos planares. Estos son raros.

• ♠ **Las espadas negras** indican problemas que requieren una cantidad significativa de trabajo pesado y/o codificación. Estos son raros.

• Æ **Las estrellas naranjas** indican que estás comiendo Lucky Charms que fueron fabricados antes de 1998. Puaj.

Estos ejercicios están diseñados como oportunidades para practicar, no como objetivos por sí mismos. El objetivo de cada problema no es resolver ese problema específico, sino ejercitar un cierto conjunto de habilidades, o practicar resolver un cierto tipo de problema. Parcialmente por esta razón, no proporciono soluciones a los ejercicios; las soluciones no son el punto. En particular, no hay "manual del instructor"; si no puedes resolver un problema tú mismo, probablemente no deberías asignárselo a tus estudiantes. Dicho esto, probablemente puedas encontrar soluciones a los problemas de tarea que he asignado este semestre en la página web del curso que esté enseñando. ¡Y nada te impide escribir un manual del instructor!

**¡Roba Este Libro!**

Este libro está publicado bajo una Licencia Creative Commons que te permite usar, redistribuir, adaptar y remezclar su contenido sin mi permiso, siempre que señales de vuelta a la fuente original. Una versión electrónica completa de este libro está disponible gratuitamente en cualquiera de las siguientes ubicaciones:

• El sitio web del libro: http://jeffe.cs.illinois.edu/teaching/algorithms/ • El atajo mnemónico: http://algorithms.wtf • El sitio de reportes de errores: https://github.com/jeffgerickson/algorithms • El Internet Archive: https://archive.org/details/Algorithms-Jeff-Erickson

El sitio web del libro también contiene varios cientos de páginas de notas de conferencia adicionales sobre material relacionado y más avanzado, así como un archivo casi completo de tareas pasadas, exámenes, problemas de discusión/laboratorio, y otros recursos de enseñanza. Siempre que enseño una clase de algoritmos, reviso, actualizo, y a veces descarto mis materiales de enseñanza, así que puedes encontrar revisiones más recientes en la página web del curso que esté enseñando actualmente.

Ya seas estudiante o instructor, eres más que bienvenido a usar cualquier subconjunto de este libro de texto o mis otras notas de conferencia en tus propias clases, sin pedir mi permiso—¡para eso las puse en la web! Sin embargo, por favor también cita este libro, ya sea por nombre o con un enlace de vuelta a http://algorithms.wtf; esto es especialmente importante si eres estudiante, y usas mis materiales de curso para ayudar con tu tarea. (Por favor también consulta con tu instructor.)

Sin embargo, si eres instructor, te animo fuertemente a complementar estos con material adicional que escribas tú mismo. Escribir el material tú mismo fortalecerá tu dominio y presentación en clase del material, lo que a su vez mejorará el dominio del material de tus estudiantes. También te ayudará a superar la frustración de lidiar con las partes de este libro que no te gustan. Todos los libros de texto son basura imperfectos, y este no es excepción.

Finalmente, por favor haz que lo que escribas esté disponible gratuitamente, fácilmente y globalmente en la web abierta—no escondido detrás de las puertas de un sistema de gestión de aprendizaje o algún otro tipo de paywall—para que estudiantes e instructores en otros lugares puedan beneficiarse de tus perspectivas únicas. En particular, si desarrollas recursos útiles que complementen directamente este libro de texto, como diapositivas, videos, o manuales de soluciones, por favor házmelo saber para que pueda agregar enlaces a tus recursos desde el sitio web del libro.

**Agradecimientos**

Este libro de texto se basa fuertemente en las contribuciones de innumerables estudiantes, profesores e investigadores de algoritmos. En particular, estoy inmensamente agradecido con más de tres mil estudiantes de Illinois que han usado mis notas de conferencia como referencia primaria, ofrecido críticas útiles (si a veces dolorosas), y sufrido a través de algunos primeros borradores verdaderamente terribles. Gracias también a muchos colegas y estudiantes alrededor del mundo que han usado estas notas en sus propias clases y han enviado retroalimentación útil y reportes de errores.

Estoy particularmente agradecido por la retroalimentación y contribuciones (especialmente ejercicios) de mis increíbles asistentes de enseñanza:

Aditya Ramani, Akash Gautam, Alex Steiger, Alina Ene, Amir Nayyeri, Asha Seetharam, Ashish Vulimiri, Ben Moseley, Brad Sturt, Brian Ensink, Chao Xu, Charlie Carlson, Chris Neihengen, Connor Clark, Dan Bullok, Dan Cranston, Daniel Khashabi, David Morrison, Ekta Manaktala, Erin Wolf Chambers, Gail Steitz, Gio Kao, Grant Czajkowski, Hsien-Chih Chang, Igor Gammer, Jacob Laurel, John Lee, Johnathon Fischer, Junqing Deng, Kent Quanrud, Kevin Milans, Kevin Small, Konstantinos Koiliaris, Kyle Fox, Kyle Jao, Lan Chen, Mark Idleman, Michael Bond, Mitch Harris, Naveen Arivazhagen, Nick Bachmair, Nick Hurlburt, Nirman Kumar, Nitish Korula, Patrick Lin, Phillip Shih, Rachit Agarwal, Reza Zamani-Nasab, Rishi Talreja, Rob McCann, Sahand Mozaffari, Shalan Naqvi, Shripad Thite, Spencer Gordon, Srihita Vatsavaya, Subhro Roy, Tana Wattanawaroon, Umang Mathur, Vipul Goyal, Yasu Furakawa, y Yipu Wang.

También he sido ayudado tremendamente por muchas discusiones con colegas de la facultad en Illinois: Alexandra Kolla, Cinda Heeren, Edgar Ramos, Herbert Edelsbrunner, Jason Zych, Kim Whittlesey, Lenny Pitt, Madhu Parasarathy, Mahesh Viswanathan, Margaret Fleck, Shang-Hua Teng, Steve LaValle, y especialmente Chandra Chekuri, Ed Reingold, y Sariel Har-Peled.

Por supuesto este libro debe una gran deuda a las personas que me enseñaron estas cosas de algoritmos en primer lugar: Bob Bixby y Michael Pearlman en Rice; David Eppstein, Dan Hirschberg, y George Lueker en Irvine; y Abhiram Ranade, Dick Karp, Manuel Blum, Mike Luby, y Raimund Seidel en Berkeley.

Robé la primera iteración de la estructura general del curso, y la idea de escribir mis propias notas de conferencia en primer lugar, de Herbert Edelsbrunner; la idea de convertir un subconjunto de mis notas en un libro de Steve LaValle; y varios componentes del diseño del libro de Robert Ghrist.

**¡Caveat Lector!**

Por supuesto, ninguna de esas personas debería ser culpada por cualquier defecto en el libro resultante. A pesar de muchas rondas de revisión y edición, este libro contiene varios errores, bugs, metidas de pata, omisiones, enredos, kludges, errores tipográficos, errores matemáticos, errores gramaticales, errores de pensamiento, pedos mentales, decisiones de diseño pobres, inexactitudes históricas, anacronismos, inconsistencias, exageraciones, vacilaciones, palabrería, distorsiones, simplificaciones excesivas, redundancia, logorrea, disparates, basura, cruft, chatarra, y mentiras descaradas, todas las cuales son enteramente culpa de Steve Skiena.

Mantengo un rastreador de problemas en https://github.com/jeffgerickson/algorithms, donde lectores como tú pueden enviar reportes de errores, solicitudes de características, y retroalimentación general sobre el libro. Por favor házmelo saber si encuentras un error de cualquier tipo, ya sea matemático, gramatical, histórico, tipográfico, cultural, o de otro tipo, ya sea en el texto principal, en los ejercicios, o en mis otros materiales de curso. (Es improbable que a Steve le importe.) ¡Por supuesto, toda otra retroalimentación también es bienvenida!

¡Disfruta!

— Jeff

*Es tradicional para el autor aceptar magnánimamente la culpa por cualesquiera deficiencias que permanezcan. Yo no lo hago. Cualesquiera errores, deficiencias, o problemas en este libro son culpa de alguien más, pero apreciaría saber sobre ellos para determinar quién tiene la culpa.*  
— Steven S. Skiena, *The Algorithm Design Manual* (1997)

*Sin duda esta declaración será seguida por una lista anotada de todos los libros de texto, y por qué cada uno es basura.*  
— Adam Contini, MetaFilter, 4 de enero de 2010

**Tabla de Contenidos**

**Prefacio** i  
Acerca de Este Libro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . i  
Prerrequisitos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . i  
Referencias Adicionales . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii  
Acerca de los Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iv  
¡Roba Este Libro! . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . v  
Agradecimientos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vi  
¡Caveat Lector! . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vii

**Tabla de Contenidos** ix

**0 Introducción** 1  
0.1 ¿Qué es un algoritmo? . . . . . . . . . . . . . . . . . . . . . . . . . . 1  
0.2 Multiplicación . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3  
*Multiplicación en Celosía • Duplicación y Mediación • Compás y Regla*  
0.3 Distribución Congresional . . . . . . . . . . . . . . . . . . . . . . . 8  
0.4 Un Mal Ejemplo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  
0.5 Descripción de Algoritmos . . . . . . . . . . . . . . . . . . . . . . . . 11  
*Especificación del Problema • Descripción del Algoritmo*  
0.6 Análisis de Algoritmos . . . . . . . . . . . . . . . . . . . . . . . . . . 14  
*Corrección • Tiempo de Ejecución*  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**1 Recursión** 21  
1.1 Reducciones . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21  
1.2 Simplificar y Delegar . . . . . . . . . . . . . . . . . . . . . . . . . . . 22  
1.3 Torre de Hanoi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24  
1.4 Mergesort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26  
*Corrección • Análisis*  
1.5 Quicksort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29  
*Corrección • Análisis*  
1.6 El Patrón . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31  
1.7 Árboles de Recursión . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31  
♥*Ignorar Pisos y Techos Está Bien, En Serio*  
1.8 ♥Selección en Tiempo Lineal . . . . . . . . . . . . . . . . . . . . . . . 35  
*Quickselect • Buenos pivotes • Análisis • Verificación de Cordura*  
1.9 Multiplicación Rápida . . . . . . . . . . . . . . . . . . . . . . . . . . . 40  
1.10 Exponenciación . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**2 Backtracking** 71  
2.1 N Reinas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71  
2.2 Árboles de Juego . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74  
2.3 Suma de Subconjuntos . . . . . . . . . . . . . . . . . . . . . . . . . . 76  
*Corrección • Análisis • Variantes*  
2.4 El Patrón General . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79  
2.5 Segmentación de Texto (Interpunctio Verborum) . . . . . . . . . . . 80  
*Formulación de Índice • ♥Análisis • Variantes*  
2.6 Subsecuencia Creciente Más Larga . . . . . . . . . . . . . . . . . . . 86  
2.7 Subsecuencia Creciente Más Larga, Segundo Intento . . . . . . . . . 89  
2.8 Árboles de Búsqueda Binaria Óptimos . . . . . . . . . . . . . . . . . 91  
♥*Análisis*  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

**3 Programación Dinámica** 97  
3.1 Matr̄avr̄tā . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97  
*El Backtracking Puede Ser Lento • Memo(r)ización: Recordar Todo • Programación Dinámica: Llenar Deliberadamente • No Recordar Todo Después de Todo*  
3.2 ♥Aparte: Números de Fibonacci Aún Más Rápidos . . . . . . . . . . 103  
*¡Whoa! ¡No tan rápido!*  
3.3 Interpunctio Verborum Redux . . . . . . . . . . . . . . . . . . . . . . 105  
3.4 El Patrón: Recursión Inteligente . . . . . . . . . . . . . . . . . . . . 105  
3.5 Advertencia: La Codicia es Estúpida . . . . . . . . . . . . . . . . . . 107  
3.6 Subsecuencia Creciente Más Larga . . . . . . . . . . . . . . . . . . . 109  
*Primera Recurrencia: ¿Es Este el Siguiente? • Segunda Recurrencia: ¿Qué Sigue?*  
3.7 Distancia de Edición . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111  
*Estructura Recursiva • Recurrencia • Programación Dinámica*  
3.8 Suma de Subconjuntos . . . . . . . . . . . . . . . . . . . . . . . . . . 116  
3.9 Árboles de Búsqueda Binaria Óptimos . . . . . . . . . . . . . . . . . 117  
3.10 Programación Dinámica en Árboles . . . . . . . . . . . . . . . . . . 120  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123

**4 Algoritmos Codiciosos** 159  
4.1 Almacenar Archivos en Cinta . . . . . . . . . . . . . . . . . . . . . . 159  
4.2 Programar Clases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161  
4.3 Patrón General . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 164  
4.4 Códigos de Huffman . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165  
4.5 Emparejamiento Estable . . . . . . . . . . . . . . . . . . . . . . . . . 170  
*Algunas Malas Ideas • Los Algoritmos Boston Pool y Gale-Shapley • Tiempo de Ejecución • Corrección • ¡Optimalidad!*  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 176

**5 Algoritmos Básicos de Grafos** 187  
5.1 Introducción e Historia . . . . . . . . . . . . . . . . . . . . . . . . . 187  
5.2 Definiciones Básicas . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190  
5.3 Representaciones y Ejemplos . . . . . . . . . . . . . . . . . . . . . . 192  
5.4 Estructuras de Datos . . . . . . . . . . . . . . . . . . . . . . . . . . . 195  
*Listas de Adyacencia • Matrices de Adyacencia • Comparación*  
5.5 Búsqueda Lo-Que-Sea-Primero . . . . . . . . . . . . . . . . . . . . . 199  
*Análisis*  
5.6 Variantes Importantes . . . . . . . . . . . . . . . . . . . . . . . . . . 201  
*Pila: Profundidad-Primero • Cola: Anchura-Primero • Cola de Prioridad: Mejor-Primero • Grafos Desconectados • Grafos Dirigidos*  
5.7 Reducciones de Grafos: Relleno de Inundación . . . . . . . . . . . . 205  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 207

**6 Búsqueda en Profundidad** 225  
6.1 Preorden y Postorden . . . . . . . . . . . . . . . . . . . . . . . . . . 227  
*Clasificación de Vértices y Aristas*  
6.2 Detección de Ciclos . . . . . . . . . . . . . . . . . . . . . . . . . . . . 231  
6.3 Ordenamiento Topológico . . . . . . . . . . . . . . . . . . . . . . . . 232  
*Ordenamiento Topológico Implícito*  
6.4 Memoización y Programación Dinámica . . . . . . . . . . . . . . . 234  
*Programación Dinámica en Dags*  
6.5 Conectividad Fuerte . . . . . . . . . . . . . . . . . . . . . . . . . . . 237  
6.6 Componentes Fuertes en Tiempo Lineal . . . . . . . . . . . . . . . . 238  
*Algoritmo de Kosaraju y Sharir • ♥Algoritmo de Tarjan*  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 244

**7 Árboles de Expansión Mínima** 257  
7.1 Pesos de Aristas Distintos . . . . . . . . . . . . . . . . . . . . . . . . 257  
7.2 El Único Algoritmo de Árbol de Expansión Mínima . . . . . . . . . 259  
7.3 Algoritmo de Borůvka . . . . . . . . . . . . . . . . . . . . . . . . . . 261  
*Este es el Algoritmo MST que Quieres*  
7.4 Algoritmo de Jarník ("Prim") . . . . . . . . . . . . . . . . . . . . . . 263  
♥*Mejorando el Algoritmo de Jarník*  
7.5 Algoritmo de Kruskal . . . . . . . . . . . . . . . . . . . . . . . . . . . 265  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 268

**8 Caminos Más Cortos** 273  
8.1 Árboles de Caminos Más Cortos . . . . . . . . . . . . . . . . . . . . 274  
8.2 ♥Aristas Negativas . . . . . . . . . . . . . . . . . . . . . . . . . . . . 274  
8.3 El Único Algoritmo SSSP . . . . . . . . . . . . . . . . . . . . . . . . 276  
8.4 Grafos No Ponderados: Búsqueda en Anchura . . . . . . . . . . . . 278  
8.5 Grafos Acíclicos Dirigidos: Búsqueda en Profundidad . . . . . . . . 282  
8.6 Mejor-Primero: Algoritmo de Dijkstra . . . . . . . . . . . . . . . . . 284  
*Sin Aristas Negativas • ♥Aristas Negativas*  
8.7 Relajar TODAS las Aristas: Bellman-Ford . . . . . . . . . . . . . . . 289  
*Mejora de Moore • Formulación de Programación Dinámica*  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 297

**9 Caminos Más Cortos Entre Todos los Pares** 309  
9.1 Introducción . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 309  
9.2 Muchas Fuentes Únicas . . . . . . . . . . . . . . . . . . . . . . . . . 310  
9.3 Reponderación . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 311  
9.4 Algoritmo de Johnson . . . . . . . . . . . . . . . . . . . . . . . . . . 312  
9.5 Programación Dinámica . . . . . . . . . . . . . . . . . . . . . . . . . 313  
9.6 Divide y Vencerás . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 315  
9.7 Multiplicación de Matrices Divertida . . . . . . . . . . . . . . . . . . 316  
9.8 (Kleene-Roy-)Floyd-Warshall(-Ingerman) . . . . . . . . . . . . . . 318  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 320

**10 Flujos Máximos y Cortes Mínimos** 327  
10.1 Flujos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 328  
10.2 Cortes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 329  
10.3 El Teorema Flujo-Máximo-Corte-Mínimo . . . . . . . . . . . . . . 331  
10.4 Algoritmo de caminos aumentantes de Ford y Fulkerson . . . . . . 334  
♥*Capacidades Irracionales*  
10.5 Combinación y Descomposición de Flujos . . . . . . . . . . . . . . 336  
10.6 Algoritmos de Edmonds y Karp . . . . . . . . . . . . . . . . . . . . 340  
*Caminos Aumentantes Más Gordos • Caminos Aumentantes Más Cortos*  
10.7 Progreso Adicional . . . . . . . . . . . . . . . . . . . . . . . . . . . 343  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 344

**11 Aplicaciones de Flujos y Cortes** 353  
11.1 Caminos Disjuntos en Aristas . . . . . . . . . . . . . . . . . . . . . . 353  
11.2 Capacidades de Vértices y Caminos Disjuntos en Vértices . . . . . . 354  
11.3 Emparejamiento Bipartito . . . . . . . . . . . . . . . . . . . . . . . . 355  
11.4 Selección de Tuplas . . . . . . . . . . . . . . . . . . . . . . . . . . . . 357  
*Programación de Exámenes*  
11.5 Cubiertas de Caminos Disjuntos . . . . . . . . . . . . . . . . . . . . 360  
*Contratación Mínima de Facultad*  
11.6 Eliminación de Béisbol . . . . . . . . . . . . . . . . . . . . . . . . . . 363  
11.7 Selección de Proyectos . . . . . . . . . . . . . . . . . . . . . . . . . . 366  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 368

**12 NP-Dureza** 379  
12.1 Un Juego que No Puedes Ganar . . . . . . . . . . . . . . . . . . . . 379  
12.2 P versus NP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 381  
12.3 NP-duro, NP-fácil, y NP-completo . . . . . . . . . . . . . . . . . . . 382  
12.4 ♥Definiciones Formales (HC SVNT DRACONES) . . . . . . . . . . 384  
12.5 Reducciones y Sat . . . . . . . . . . . . . . . . . . . . . . . . . . . . 385  
12.6 3Sat (desde CircuitSat) . . . . . . . . . . . . . . . . . . . . . . . . 388  
12.7 Conjunto Independiente Máximo (desde 3Sat) . . . . . . . . . . . 390  
12.8 El Patrón General . . . . . . . . . . . . . . . . . . . . . . . . . . . . 392  
12.9 Clique y Cubierta de Vértices (desde Conjunto Independiente) . . . 394  
12.10 Coloreo de Grafos (desde 3Sat) . . . . . . . . . . . . . . . . . . . . 395  
12.11 Ciclo Hamiltoniano . . . . . . . . . . . . . . . . . . . . . . . . . . . 398  
*Desde Cubierta de Vértices • Desde 3Sat • Variantes y Extensiones*  
12.12 Suma de Subconjuntos (desde Cubierta de Vértices) . . . . . . . . 402  
*¡Caveat Reductor!*  
12.13 Otros Problemas NP-duros Útiles . . . . . . . . . . . . . . . . . . . 404  
12.14 Eligiendo el Problema Correcto . . . . . . . . . . . . . . . . . . . . 407  
12.15 Un Ejemplo Frívolo del Mundo Real . . . . . . . . . . . . . . . . . . 408  
12.16 ♥Más Allá de Zebra . . . . . . . . . . . . . . . . . . . . . . . . . . . 412  
*Espacio Polinomial • Tiempo Exponencial • ¡Excelsior!*  
Ejercicios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 415

**Índice** 442  
**Índice de Personas** 446  
**Índice de Pseudocódigo** 449  
**Créditos de Imágenes** 451  
**Colofón** 453

*Hinc incipit algorismus. Haec algorismus ars praesens dicitur in qua talibus indorum fruimur bis quinque figuris 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.*  
— Frater Alexander de Villa Dei, *Carmen de Algorismo* (c. 1220)

*Tienes razón en exigir que un artista se comprometa conscientemente con su trabajo, pero confundes dos cosas diferentes: resolver el problema y plantear correctamente la pregunta.*  
— Anton Chekhov, en una carta a A. S. Suvorin (27 de octubre de 1888)

*Cuanto más nos reducimos a máquinas en las cosas menores, más fuerza liberaremos para usar en las superiores.*  
— Anna C. Brackett, *The Technique of Rest* (1892)

*Y aquí estoy a las 2:30 a.m. escribiendo sobre técnica, a pesar de una fuerte convicción de que en el momento en que un hombre comienza a hablar sobre técnica, eso es prueba de que se quedó sin ideas.*  
— Raymond Chandler, carta a Erle Stanley Gardner (5 de mayo de 1939)

*Los hombres buenos no necesitan reglas. Hoy no es el día para descubrir por qué tengo tantas.*  
— El Doctor [Matt Smith], "A Good Man Goes to War", *Doctor Who* (2011)

**0 Introducción**

**0.1 ¿Qué es un algoritmo?**

Un algoritmo es una secuencia explícita, precisa, sin ambigüedades, ejecutable mecánicamente de instrucciones elementales, usualmente destinada a lograr un propósito específico. Por ejemplo, aquí está un algoritmo para cantar esa canción molesta "99 Botellas de Cerveza en la Pared", para valores arbitrarios de 99:

BotellaseDeCerveza(n):

Para i ← n hacia abajo hasta 1

Cantar "i botellas de cerveza en la pared, i botellas de cerveza,"

Cantar "Toma una, pásala, i − 1 botellas de cerveza en la pared."

Cantar "No hay botellas de cerveza en la pared, no hay botellas de cerveza,"

Cantar "Ve a la tienda, compra algunas más, n botellas de cerveza en la pared."

La palabra "algoritmo" no deriva, como podrían suponer los clasicistas algoritmofóbicos, de las raíces griegas arithmos (άριθμός), que significa "número", y algos (ἄλγος), que significa "dolor". Más bien, es una corrupción del nombre del erudito persa del siglo IX Muḥammad ibn Mūsā al-Khwārizmī.¹ Al-Khwārizmī es quizás mejor conocido como el escritor del tratado *Al-Kitab al-mukhtaṣar fī ḥisāb al-ğabr wa'l-muqābalah*,² del cual deriva la palabra moderna álgebra. En un tratado diferente, al-Khwārizmī describió el sistema decimal moderno para escribir y manipular números—en particular, el uso de un pequeño círculo o ṣifr para representar una cantidad faltante—que había sido desarrollado en India varios siglos antes.

Los métodos descritos en este último tratado, usando ya sea figuras escritas o piedras de contar, se conocieron en inglés como algorism o augrym, y sus figuras se conocieron en inglés como ciphers.

Aunque tanto la notación posicional como las obras de al-Khwārizmī ya eran conocidas por algunos eruditos europeos, el sistema numérico "hindu-árabe" fue popularizado en Europa por el matemático y comerciante italiano medieval Leonardo de Pisa, mejor conocido como Fibonacci. Gracias en parte a su libro de 1202 *Liber Abaci*,³ las figuras escritas comenzaron a reemplazar la tabla de contar (entonces conocida como ábaco) y la aritmética con dedos⁴ como la plataforma preferida para el cálculo⁵ en Europa en el siglo XIII—no porque las figuras decimales escritas fueran más fáciles de aprender o usar, sino porque proporcionaban una pista de auditoría. Los cifras se volvieron comunes en Europa Occidental sólo con el advenimiento de los tipos móviles, y verdaderamente ubicuos sólo después de que el papel barato se volviera abundante a principios del siglo XIX.

Eventualmente la palabra algorism evolucionó al algoritmo moderno, vía etimología popular del griego arithmos (y quizás el algos mencionado anteriormente).⁶ Así, hasta muy recientemente, la palabra algoritmo se refería exclusivamente a técnicas mecánicas para aritmética de valor posicional usando numerales "árabes". Las personas entrenadas en la ejecución rápida y confiable de estos procedimientos se llamaban algoristas o computadores, o más simplemente, computadoras.

¹ "Mohammad, padre de Adbdulla, hijo de Moisés, el Kwarizmiano". Kwārizm es una ciudad antigua, ahora llamada Khiva, en la Provincia de Khorezm de Uzbekistán.

² "El Libro Compendioso sobre Cálculo por Completación y Balanceado"

³ Mientras es tentador traducir el título *Liber Abaci* como "El Libro del Ábaco", una traducción más precisa es "El Libro del Cálculo". Tanto antes como después de Fibonacci, la palabra italiana abaco se usaba para describir cualquier cosa relacionada con el cálculo numérico—dispositivos, métodos, escuelas, libros, etc.—de la misma manera que "ciencias de la computación" se usa hoy en inglés, o como la frase china para "investigación de operaciones" se traduce literalmente como "el estudio de usar varillas de contar".

⁴ ☞ ¡Calcular con dígitos! ☞

⁵ La palabra calcular deriva de la palabra latina calculus, que significa "piedra pequeña", refiriéndose a las piedras en una tabla de contar, o como las llamaba Chaucer, piedras augrym. En 440 a.C., Heródoto escribió en sus *Historias* que "Los griegos escriben y calculan (λογίζεσθαι ψήφοις, literalmente 'calculan con guijarros') de izquierda a derecha; los egipcios hacen lo contrario. Sin embargo, dicen que su forma de escribir es hacia la derecha, y la forma griega hacia la izquierda." (Heródoto permanece extrañamente silencioso sobre qué extremo del huevo comían primero los egipcios.)

⁶ Algunas fuentes medievales afirman que el prefijo griego "algo-" significa "arte" o "introducción". Otras afirman que los algoritmos fueron inventados por un filósofo griego, o un rey de India, o quizás un rey de España, llamado "Algus" o "Algor" o "Argus". Algunas, posiblemente incluyendo a Dante Alighieri, incluso identificaron al inventor con el constructor de barcos mitológico griego y argonauta epónimo. No está claro si alguna de estas afirmaciones risibles pretendía ser históricamente precisa, o meramente mnemónica.

**0.2 Multiplicación**

Aunque han sido un tema de estudio académico formal por sólo unas pocas décadas, los algoritmos han estado con nosotros desde el amanecer de la civilización. Las descripciones de computación aritmética paso a paso están entre los primeros ejemplos de lenguaje humano escrito, precediendo por mucho las exposiciones de Fibonacci y al-Khwārizmī, o incluso la notación posicional que popularizaron.

**Multiplicación en Celosía**

El método más familiar para multiplicar números grandes, al menos para estudiantes estadounidenses, es el algoritmo de celosía. Este algoritmo fue popularizado por Fibonacci en *Liber Abaci*, quien lo aprendió de fuentes árabes incluyendo al-Khwārizmī, quien a su vez lo aprendió de fuentes indias incluyendo el tratado del siglo VII *Brahmasphuṭasiddhāntā* de Brahmagupta, quien pudo haberlo aprendido de fuentes chinas. Las descripciones más antiguas sobrevivientes del algoritmo aparecen en *El Clásico Matemático de Sunzi*, escrito en China entre los siglos III y V, y en los comentarios de Eutocius de Ascalón sobre *La Medida del Círculo* de Arquímedes, escritos alrededor del 500 d.C., pero hay evidencia de que el algoritmo era conocido mucho antes. Eutocius acredita el método a un tratado perdido de Apolonio de Perga, quien vivió alrededor del 300 a.C., titulado *Okytokion* (᾿Ωκυτόκιον).⁷ Los sumerios registraron tablas de multiplicación en tabletas de arcilla tan temprano como el 2600 a.C., sugiriendo que podrían haber usado el algoritmo de celosía.⁸

El algoritmo de celosía asume que los números de entrada están representados como cadenas explícitas de dígitos; asumiré aquí que estamos trabajando en base diez, pero el algoritmo se generaliza inmediatamente a cualquier otra base. Para simplificar la notación,⁹ la entrada consiste en un par de arreglos X[0 .. m − 1] y Y[0 .. n − 1], representando los números

x = ∑(i=0 a m-1) X[i] · 10^i y y = ∑(j=0 a n-1) Y[j] · 10^j,

y similarmente, la salida consiste en un solo arreglo Z[0 .. m + n−1], representando el producto

z = x · y = ∑(k=0 a m+n-1) Z[k] · 10^k.

El algoritmo usa adición y multiplicación de un solo dígito como operaciones primitivas. La adición puede realizarse usando un bucle for simple. En la práctica, la multiplicación de un solo dígito se realiza usando una tabla de búsqueda, ya sea grabada en tabletas de arcilla, pintada en tiras de madera o bambú, escrita en papel, almacenada en memoria de solo lectura, o memorizada por el computador. Todo el algoritmo de celosía puede resumirse por la fórmula

x · y = ∑(i=0 a m-1) ∑(j=0 a n-1) [X[i] · Y[j] · 10^(i+j)]

Diferentes variantes del algoritmo de celosía evalúan los productos parciales X[i] · Y[j] · 10^(i+j) en diferentes órdenes y usan diferentes estrategias para computar su suma. Por ejemplo, en *Liber Abaco*, Fibonacci describe una variante que considera los mn productos parciales en orden creciente de significancia, como se muestra en pseudocódigo moderno abajo.

FibonacciMultiply(X[0 .. m − 1], Y[0 .. n − 1]):

hold ← 0

para k ← 0 hasta n + m − 1

para todo i y j tal que i + j = k

hold ← hold + X[i] · Y[j]

Z[k] ← hold mod 10

hold ← ⌊hold/10⌋

retornar Z[0 .. m + n − 1]

El algoritmo de Fibonacci a menudo se ejecuta almacenando todos los productos parciales en una tabla bidimensional (a menudo llamada "tableau" o "rejilla" o "celosía") y luego sumando a lo largo de las diagonales con acarreos apropiados, como se muestra a la derecha en la Figura 0.1. A los estudiantes de primaria estadounidenses se les enseña a multiplicar un factor (el "multiplicando") por cada dígito en el otro factor (el "multiplicador"), escribiendo todos los productos multiplicando-por-dígito antes de sumarlos, como se muestra a la izquierda en la Figura 0.1. Este también fue el método descrito por Eutocius, aunque él apropiadamente consideró los dígitos del multiplicador de izquierda a derecha, como se muestra en la Figura 0.2. Ambas variantes (y varias otras) están descritas e ilustradas lado a lado en el libro de texto anónimo de 1458 *L'Arte dell'Abbaco*, también conocido como la Aritmética de Treviso, el primer libro de matemáticas impreso en Occidente.

**Figura 0.1.** Calculando 934 × 314 = 293276 usando multiplicación "larga" (con verificación de errores desechando nueves) y multiplicación de "celosía", de *L'Arte dell'Abbaco* (1458). (Ver Créditos de Imágenes al final del libro.)

**Figura 0.2.** Cálculo del siglo VI de Eutocius de 1172⅛ × 1172⅛ = 1373877 1/64, en su comentario sobre *La Medida del Círculo* de Arquímedes, transcrito (izquierda) y traducido a notación moderna (derecha) por Johan Heiberg (1891). (Ver Créditos de Imágenes al final del libro.)

Todas estas variantes del algoritmo de celosía—y otras variantes similares descritas por Sunzi, al-Khwārizmī, Fibonacci, *L'Arte dell'Abbaco*, y muchas otras fuentes—calculan el producto de cualquier número de m dígitos y cualquier número de n dígitos en tiempo O(mn); el tiempo de ejecución de cada variante está dominado por el número de multiplicaciones de un solo dígito.

**Duplicación y Mediación**

El algoritmo de celosía no es el algoritmo de multiplicación más antiguo para el cual tenemos evidencia registrada directa. Un algoritmo incluso más antiguo y posiblemente más simple, que no depende de la notación posicional, a veces se llama multiplicación del campesino ruso, multiplicación del campesino etíope, o simplemente multiplicación del campesino.

⁷ ¡Literalmente "medicina que promueve el parto rápido y fácil"! Papus de Alejandría reprodujo varios extractos de *Okytokion* unos 200 años antes que Eutocius, pero su descripción del algoritmo de multiplicación de celosía (si la dio) también se perdió.

⁸ Hay amplia evidencia de que los antiguos sumerios calculaban precisamente con números extremadamente grandes usando su sistema numérico posicional base-60, pero no estoy consciente de ningún registro sobreviviente de los métodos reales que usaban. Además de las tablas estándar de multiplicación y recíprocos, se han encontrado tablas listando los cuadrados de enteros del 1 al 59, llevando a algunos historiadores de matemáticas a conjeturar que los babilonios multiplicaban usando una identidad como xy = ((x + y)² − x² − y²)/2. Pero este truco sólo funciona cuando x + y < 60; la historia es silenciosa sobre cómo los babilonios podrían haber computado x² cuando x ≥ 60.

⁹ pero con el riesgo de inflamar la enemistad histórica entre Grecia y Egipto, o Lilliput y Blefuscu, o Macs y PCs, o personas que piensan que cero es un número natural y personas que están equivocadas

Una variante de este algoritmo fue copiada en el papiro Rhind por el escriba egipcio Ahmes alrededor del 1650 a.C., de un documento que él afirmaba era (entonces) de unos 350 años de antigüedad.¹⁰ Este algoritmo todavía se enseñaba en escuelas primarias en Europa Oriental a fines del siglo XX; también era comúnmente usado por las primeras computadoras digitales que no implementaban multiplicación de enteros directamente en hardware.

El algoritmo de multiplicación del campesino reduce la difícil tarea de multiplicar números arbitrarios a una secuencia de cuatro operaciones más simples: (1) determinar paridad (par o impar), (2) adición, (3) duplicación (doblar un número), y (4) mediación (dividir un número por la mitad, redondeando hacia abajo).

PeasantMultiply(x, y):

prod ← 0

mientras x > 0

si x es impar

prod ← prod + y

x ← ⌊x/2⌋

y ← y + y

retornar prod

| **x** | **y** | **prod** |
| --- | --- | --- |
| 123 | +456 | = 456 |
| 61 | +912 | = 1368 |
| 30 | 1824 |  |
| 15 | +3648 | = 5016 |
| 7 | +7296 | = 12312 |
| 3 | +14592 | = 26904 |
| 1 | +29184 | = 56088 |

**Figura 0.3.** Multiplicación por duplicación y mediación

La corrección de este algoritmo sigue por inducción de la siguiente identidad recursiva, que se mantiene para todos los enteros no negativos x y y:

x · y = { 0 si x = 0 { ⌊x/2⌋ · (y + y) si x es par { ⌊x/2⌋ · (y + y) + y si x es impar

Posiblemente, esta recurrencia es el algoritmo de multiplicación del campesino. ¡No dejes que el pseudocódigo iterativo te engañe; el algoritmo es fundamentalmente recursivo!

Como se establece, PeasantMultiply realiza O(log x) operaciones de paridad, adición y mediación, pero podemos mejorar esta cota a O(log min{x, y}) intercambiando los dos argumentos cuando x > y. Asumiendo que los números están representados usando cualquier notación posicional razonable (como binaria, decimal, hexagesimal babilónica, duodecimal egipcia, números romanos, varillas de contar chinas, posiciones de cuentas en un ábaco, etc.), cada operación requiere a lo más O(log(xy)) = O(log max{x, y}) operaciones de un solo dígito, así que el tiempo de ejecución total del algoritmo es O(log min{x, y} · log max{x, y}) = O(log x · log y).

En otras palabras, este algoritmo requiere tiempo O(mn) para multiplicar un número de m dígitos por un número de n dígitos; hasta factores constantes, este es el mismo tiempo de ejecución que el algoritmo de celosía. Este algoritmo requiere (¡un factor constante!) más papeleo para ejecutar a mano que el algoritmo de celosía, pero las operaciones primitivas necesarias son posiblemente más fáciles para los humanos de realizar. De hecho, los dos algoritmos son equivalentes cuando los números están representados en binario.

**Compás y Regla**

Los geómetras griegos clásicos identificaron números (o más precisamente, magnitudes) con segmentos de línea de la longitud apropiada, que manipulaban usando dos herramientas mecánicas simples—el compás y la regla—versiones de las cuales ya habían estado en uso común por topógrafos, arquitectos y otros artesanos durante siglos. Usando sólo estas dos herramientas, estos eruditos redujeron varias construcciones geométricas complejas a las siguientes operaciones primitivas, comenzando con uno o más puntos de referencia identificados.

• Dibujar la línea única que pasa por dos puntos identificados distintos. • Dibujar el círculo único centrado en un punto identificado y pasando por otro. • Identificar el punto de intersección (si existe) de dos líneas. • Identificar los puntos de intersección (si existen) de una línea y un círculo. • Identificar los puntos de intersección (si existen) de dos círculos.

En la práctica, los estudiantes de geometría griega casi ciertamente dibujaron sus construcciones en un abax (ἄβαξ), una mesa cubierta de polvo o arena.¹¹ Siglos antes, los topógrafos egipcios llevaron a cabo muchas de las mismas construcciones usando cuerdas para determinar líneas rectas y círculos en el suelo.¹² Sin embargo, Euclides y otros geómetras griegos presentaron las construcciones de compás y regla como abstracciones matemáticas precisas—los puntos son puntos ideales; las líneas son líneas ideales; y los círculos son círculos ideales.

La Figura 0.4 muestra un algoritmo, descrito en los *Elementos* de Euclides hace unos 2500 años, para multiplicar o dividir dos magnitudes. La entrada consiste en cuatro puntos distintos A, B, C y D, y el objetivo es construir un punto Z tal que |AZ| = |AC||AD|/|AB|. En particular, si definimos |AB| como nuestra unidad de longitud, entonces el algoritmo calcula el producto de |AC| y |AD|.

Nota que Euclides primero define una nueva operación primitiva RightAngle (como los programadores modernos lo frasearían) escribiendo una subrutina. La corrección del algoritmo sigue de la observación de que los triángulos ACE y AZF son similares. La segunda y tercera líneas del algoritmo principal son ambiguas, porque α intersecta cualquier círculo centrado en A en dos puntos distintos, pero el algoritmo es realmente correcto sin importar qué puntos de intersección se elijan para E y F.

⟨⟨Construir la línea perpendicular a ℓ pasando por P.⟩⟩

RightAngle(ℓ, P):

Elegir un punto A ∈ ℓ

A, B ← Intersect(Circle(P,A), ℓ)

C, D ← Intersect(Circle(A, B),Circle(B,A))

retornar Line(C, D)

⟨⟨Construir un punto Z tal que |AZ| = |AC||AD|/|AB|.⟩⟩

MultiplyOrDivide(A, B, C, D):

α ← RightAngle(Line(A, C),A)

E ← Intersect(Circle(A, B),α)

F ← Intersect(Circle(A, D),α)

β ← RightAngle(Line(E, C), F)

γ ← RightAngle(β, F)

retornar Intersect(γ, Line(A, C))

**Figura 0.4.** Multiplicación por compás y regla.

El algoritmo de Euclides reduce el problema de multiplicar dos magnitudes (longitudes) a una serie de operaciones primitivas de compás y regla. Estas operaciones son difíciles de implementar precisamente en una computadora digital moderna, pero el algoritmo de Euclides no fue diseñado para una computadora digital. Fue diseñado para el Geómetra Ideal Platónico, empuñando el Compás Ideal Platónico y la Regla Ideal Platónica, quien podía ejecutar cada operación perfectamente en tiempo constante por definición. ¡En este modelo de computación, MultiplyOrDivide ejecuta en tiempo O(1)!

¹⁰ La versión de este algoritmo realmente usada en el antiguo Egipto no usa mediación o paridad, pero sí usa comparaciones. Para evitar dividir por la mitad, el algoritmo pre-calcula dos tablas por duplicación repetida: una conteniendo todas las potencias de 2 que no exceden x, la otra conteniendo las mismas potencias de 2 multiplicadas por y. Las potencias de 2 que suman x se encuentran entonces por sustracción codiciosa, y las entradas correspondientes en la otra tabla se suman para formar el producto.

¹¹ Los numerales escritos 1 hasta 9 eran conocidos en Europa al menos dos siglos antes del *Liber Abaci* de Fibonacci como "numerales gobar", de la palabra árabe ghubār que significa polvo, refiriéndose últimamente a la práctica india de realizar aritmética en mesas cubiertas con arena. La palabra griega ἄβαξ es el origen del latín abacus, que también originalmente se refería a una mesa de arena.

¹² ¿Recuerdas lo que significa "geometría"? Demócrito se referiría más tarde a estos topógrafos egipcios, de manera algo despectiva, como arpedonaptai (ἀρπεδονάπται), que significa "atadores de cuerdas".

**0.3 Distribución Congresional**

Aquí hay otro ejemplo del mundo real de un algoritmo de importante significancia política. El Artículo I, Sección 2 de la Constitución de los Estados Unidos requiere que

Los Representantes e Impuestos directos serán distribuidos entre los varios Estados que puedan estar incluidos dentro de esta Unión, de acuerdo a sus respectivos Números.... El Número de Representantes no excederá uno por cada treinta Mil, pero cada Estado tendrá al Menos un Representante....

Porque hay sólo un número finito de asientos en la Cámara de Representantes, la representación proporcional exacta requiere ya sea representantes compartidos o fraccionarios, ninguno de los cuales es legal. Como resultado, durante las siguientes décadas, muchos algoritmos de distribución diferentes fueron propuestos y usados para redondear la solución fraccionaria ideal justamente. El algoritmo realmente usado hoy, llamado el método Huntington-Hill o el método de proporciones iguales, fue primero sugerido por el estadístico del Bureau del Censo Joseph Hill en 1911, refinado por el matemático de Harvard Edward Huntington en 1920, adoptado en la ley Federal (2 U.S.C. §2a) en 1941, y sobrevivió un desafío de la Corte Suprema en 1992.¹³

El método Huntington-Hill asigna representantes a estados uno a la vez. Primero, en una etapa de preprocesamiento, cada estado es asignado un representante. Luego en cada iteración del bucle principal, el siguiente representante se asigna al estado con la prioridad más alta. La prioridad de cada estado se define como P/√(r(r + 1)), donde P es la población del estado y r es el número de representantes ya asignados a ese estado.

El algoritmo se describe en pseudocódigo en la Figura 0.5. La entrada consiste en un arreglo Pop[1 .. n] almacenando las poblaciones de los n estados y un entero R igual al número total de representantes; el algoritmo asume R ≥ n. (Actualmente, en los Estados Unidos, n = 50 y R = 435.) El arreglo de salida Rep[1 .. n] registra el número de representantes asignados a cada estado.

ApportionCongress(Pop[1 .. n],R):

PQ ← NewPriorityQueue

⟨⟨Dar a cada estado su primer representante⟩⟩

para s ← 1 hasta n

Rep[s] ← 1

Insert PQ, s, Pop[i]/√2

⟨⟨Asignar los n − R representantes restantes⟩⟩

para i ← 1 hasta n − R

s ← ExtractMax(PQ)

Rep[s] ← Rep[s] + 1

priority ← Pop[s]/√(Rep[s] (Rep[s] + 1))

Insert(PQ,s, priority)

retornar Rep[1 .. n]

**Figura 0.5.** El algoritmo de distribución Huntington-Hill

Esta implementación de Huntington-Hill usa una cola de prioridad que soporta las operaciones NewPriorityQueue, Insert, y ExtractMax. (La ley actual no dice nada sobre colas de prioridad, por supuesto.) La salida del algoritmo, y por tanto su corrección, no depende en absoluto de cómo se implementa esta cola de prioridad. El Bureau del Censo usa un arreglo ordenado, almacenado en una sola columna de una hoja de cálculo de Excel, que se recalcula desde cero en cada iteración. Tú (deberías haber) aprendido una implementación más eficiente en tu clase de estructuras de datos de pregrado.

Algoritmos de distribución similares se usan en elecciones parlamentarias multipartidarias alrededor del mundo, donde el número de asientos asignados a cada partido se supone que es proporcional al número de votos que ese partido recibe. Los dos más comunes son el método D'Hondt¹⁴ y el método Webster–Sainte-Laguë,¹⁵ que respectivamente usan prioridades P/(r + 1) y P/(2r + 1) en lugar de la expresión de raíz cuadrada en Huntington-Hill. El método Huntington-Hill es esencialmente único a la Cámara de Representantes de los Estados Unidos, gracias en parte al requisito constitucional de que cada estado debe ser asignado al menos un representante.

¹³ Anulando una decisión anterior de un tribunal federal de distrito, la Corte Suprema sostuvo unánimemente que cualquier método de distribución adoptado de buena fe por el Congreso es constitucional (*United States Department of Commerce v. Montana*). El algoritmo actual de distribución congresional se describe en detalle espantoso en el sitio web del Departamento del Censo de EE.UU. http://www.census.gov/topics/public-sector/congressional-apportionment.html. Una buena historia del problema de distribución se puede encontrar en http://www.thirty-thousand.org/pages/Apportionment.htm. Un reporte del Servicio de Investigación del Congreso describiendo varios métodos de distribución está disponible en http://www.fas.org/sgp/crs/misc/R41382.pdf.

¹⁴ desarrollado por Thomas Jefferson en 1792, usado para distribución Congresional de EE.UU. de 1792 a 1832, redescubierto por el matemático belga Victor D'Hondt en 1878, y refinado por el físico suizo Eduard Hagenbach-Bischoff en 1888.

¹⁵ desarrollado por Daniel Webster en 1832, usado para distribución Congresional de EE.UU. de 1842 a 1911, redescubierto por el matemático francés André Sainte-Laguë en 1910, y redescubierto otra vez por el físico alemán Hans Schepers en 1980.

**0.4 Un Mal Ejemplo**

Como ejemplo prototípico de una secuencia de instrucciones que no es realmente un algoritmo, considera el "algoritmo de Martin":¹⁶

BeAMillionaireAndNeverPayTaxes( ):

Consigue un millón de dólares.

Si el hombre de los impuestos viene a tu puerta y dice, "¡Nunca has pagado impuestos!"

Di "Se me olvidó."

Bastante simple, excepto por ese primer paso; ¡es tremendo! Un grupo de CEOs billonarios, capitalistas de riesgo de Silicon Valley, o hustlers de bienes raíces de la ciudad de Nueva York podrían considerar esto un algoritmo, porque para ellos el primer paso es tanto sin ambigüedad como trivial,¹⁷ pero para el resto de nosotros pobres desgraciados, el procedimiento de Martin es demasiado vago para ser considerado un algoritmo real. Por otro lado, este es un ejemplo perfecto de una reducción—reduce el problema de ser millonario y nunca pagar impuestos al problema "más fácil" de adquirir un millón de dólares. Veremos reducciones una y otra vez en este libro. Como cientos de empresarios y políticos han demostrado, si sabes cómo resolver el problema más fácil, una reducción te dice cómo resolver el más difícil.

El algoritmo de Martin, como algunos de nuestros ejemplos previos, no es el tipo de algoritmo sobre el que los científicos de la computación están acostumbrados a pensar, porque está fraseado en términos de operaciones que son difíciles para las computadoras de realizar. Este libro se enfoca (¡casi!) exclusivamente en algoritmos que pueden ser razonablemente implementados en una computadora digital estándar. Cada paso en estos algoritmos es ya sea directamente soportado por lenguajes de programación comunes (como aritmética, asignaciones, bucles, o recursión) o algo que ya has aprendido cómo hacer (como ordenamiento, búsqueda binaria, recorrido de árboles, o cantar "n Botellas de Cerveza en la Pared").

¹⁶ Steve Martin, "You Can Be A Millionaire", *Saturday Night Live*, 21 de enero de 1978. También aparece en *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

¹⁷ Algo algo blockchain cuántico seguro aprendizaje profundo algo.

**0.5 Descripción de Algoritmos**

Las habilidades requeridas para diseñar y analizar algoritmos efectivamente están entrelazadas con las habilidades requeridas para describir algoritmos efectivamente. Al menos en mis clases, una descripción completa de cualquier algoritmo tiene cuatro componentes:

• **Qué:** Una especificación precisa del problema que el algoritmo resuelve. • **Cómo:** Una descripción precisa del algoritmo mismo. • **Por qué:** Una demostración de que el algoritmo resuelve el problema que se supone que resuelve. • **Qué tan rápido:** Un análisis del tiempo de ejecución del algoritmo.

No es necesario (o incluso aconsejable) desarrollar estos cuatro componentes en este orden particular. Las especificaciones de problemas, descripciones de algoritmos, demostraciones de corrección, y análisis de tiempo usualmente evolucionan simultáneamente, con el desarrollo de cada componente informando el desarrollo de los otros. Por ejemplo, podríamos necesitar ajustar la descripción del problema para soportar un algoritmo más rápido, o modificar el algoritmo para manejar un caso complicado en la demostración de corrección. Sin embargo, presentar estos componentes por separado usualmente es lo más claro para el lector.

Como con cualquier escritura, es importante dirigir tus descripciones a la audiencia correcta; recomiendo escribir para un programador competente pero escéptico que no es tan inteligente como tú. Piensa en ti mismo hace seis meses. Mientras desarrollas cualquier algoritmo nuevo, naturalmente construirás mucha intuición sobre el problema y sobre cómo tu algoritmo lo resuelve, y tu razonamiento informal será guiado por esa intuición. Pero cualquiera que lea tu algoritmo después, o el código que derives de él, no compartirá tu intuición o experiencia. Tampoco lo hará tu compilador. Tampoco lo harás tú en seis meses. Todo lo que tendrán es tu descripción escrita.

Incluso si nunca tienes que explicar tus algoritmos a nadie más, todavía es importante desarrollarlos con una audiencia en mente. Tratar de comunicarse claramente te fuerza a pensar más claramente. En particular, escribir para una audiencia novata, que interpretará tus palabras exactamente como están escritas, te fuerza a trabajar a través de detalles finos, sin importar qué tan "obvias" o "intuitivas" puedan parecer tus ideas de alto nivel en el momento. Similarmente, escribir para una audiencia escéptica te fuerza a desarrollar argumentos robustos para corrección y eficiencia, en lugar de confiar en tu intuición o tu inteligencia.¹⁸

No puedo enfatizar este punto lo suficiente: Tu trabajo primario como diseñador de algoritmos es enseñar a otras personas cómo y por qué funcionan tus algoritmos. Si no puedes comunicar tus ideas a otros seres humanos, podrían también no existir. Producir código ejecutable correcto y eficiente es un objetivo importante pero secundario. Convencer a ti mismo, tus profesores, tus (prospectivos) empleadores, tus colegas, o tus estudiantes de que eres inteligente es en el mejor de los casos un tercero distante.

**Especificación del Problema**

Antes de que podamos siquiera empezar a desarrollar un nuevo algoritmo, tenemos que acordar sobre qué problema se supone que nuestro algoritmo resuelve. Similarmente, antes de que podamos siquiera empezar a describir un algoritmo, tenemos que describir el problema que el algoritmo se supone que resuelve.

Los problemas algorítmicos a menudo se presentan usando inglés estándar, en términos de objetos del mundo real. Es responsabilidad nuestra, los diseñadores de algoritmos, reformular estos problemas en términos de objetos matemáticos formales, abstractos—números, arreglos, listas, grafos, árboles, etc.—sobre los que podemos razonar formalmente. También debemos determinar si la declaración del problema lleva alguna suposición oculta, y declarar esas suposiciones explícitamente. (Por ejemplo, en la canción "n Botellas de Cerveza en la Pared", n es siempre un entero no negativo.¹⁹)

Podríamos necesitar refinar nuestra especificación mientras desarrollamos el algoritmo. Por ejemplo, nuestro algoritmo podría requerir una representación de entrada particular, o producir una representación de salida particular, que se dejó sin especificar en la descripción original informal del problema. O nuestro algoritmo podría realmente resolver un problema más general de lo que originalmente se nos pidió resolver. (Esta es una característica común de los algoritmos recursivos.)

La especificación debería incluir justo suficiente detalle para que alguien más pueda usar nuestro algoritmo como una caja negra, sin saber cómo o por qué el algoritmo realmente funciona. En particular, debemos describir el tipo y significado de cada parámetro de entrada, y exactamente cómo la salida eventual depende de los parámetros de entrada. Por otro lado, nuestra especificación debería deliberadamente ocultar cualquier detalle que no sea necesario para usar el algoritmo como una caja negra. Deja que lo que no importa verdaderamente se deslice.

Por ejemplo, los algoritmos de celosía y duplicación-y-mediación ambos resuelven el mismo problema: Dados dos enteros no negativos x y y, cada uno representado como un arreglo de dígitos, calcular el producto x · y, también representado como un arreglo de dígitos. Para alguien usando estos algoritmos, la elección del algoritmo es completamente irrelevante. Por otro lado, el algoritmo griego de regla y compás resuelve un problema diferente, porque los valores de entrada y salida están representados por segmentos de línea en lugar de arreglos de dígitos.

**Descripción del Algoritmo**

Los programas de computadora son representaciones concretas de algoritmos, pero los algoritmos no son programas. Más bien, los algoritmos son procedimientos mecánicos abstractos que pueden ser implementados en cualquier lenguaje de programación que soporte las operaciones primitivas subyacentes. Los detalles sintácticos idiosincráticos de tu lenguaje de programación favorito son completamente irrelevantes; enfocarse en estos sólo te distraerá (y a tus lectores) de lo que realmente está pasando.²⁰ Una buena descripción de algoritmo está más cerca de lo que deberíamos escribir en los comentarios de un programa real que el código mismo. El código es un medio pobre para contar historias.

Por otro lado, una descripción en prosa en inglés plano usualmente tampoco es una buena idea. Los algoritmos tienen mucha estructura idiomática—especialmente condicionales, bucles, llamadas a funciones, y recursión—que son demasiado fácilmente ocultas por prosa no estructurada. El inglés coloquial está lleno de ambigüedades y matices de significado, pero los algoritmos deben ser descritos tan sin ambigüedad como sea posible. La prosa es un medio pobre para la precisión.

En mi opinión, la manera más clara de presentar un algoritmo es usando una combinación de pseudocódigo e inglés estructurado. El pseudocódigo usa la estructura de lenguajes de programación formales y matemáticas para dividir algoritmos en pasos primitivos; los pasos primitivos mismos pueden escribirse usando notación matemática, inglés puro, o una mezcla apropiada de los dos, lo que sea más claro. El pseudocódigo bien escrito revela la estructura interna del algoritmo pero oculta detalles de implementación irrelevantes, haciendo el algoritmo más fácil de entender, analizar, depurar, e implementar.

Siempre que describamos un algoritmo, nuestra descripción debería incluir cada detalle necesario para especificar completamente el algoritmo, demostrar su corrección, y analizar su tiempo de ejecución. Al mismo tiempo, debería excluir cualquier detalle que no sea necesario para especificar completamente el algoritmo, demostrar su corrección, y analizar su tiempo de ejecución. (Deslizar.) A un nivel más práctico, nuestra descripción debería permitir a un programador competente pero escéptico que no ha leído este libro implementar rápida y correctamente el algoritmo en su lenguaje de programación favorito, sin entender por qué funciona.

No quiero aburrirte con las reglas que sigo para escribir pseudocódigo, pero debo advertir contra un hábito especialmente pernicioso. Nunca describas operaciones repetidas informalmente, como en "Haz [esto] primero, luego haz [eso] segundo, y así sucesivamente." o "Repite este proceso hasta [algo]". Como cualquiera que haya tomado una de esas pruebas frustrantes de "¿Qué viene después en esta secuencia?" ya sabe, describir los primeros pasos de un algoritmo dice poco o nada sobre lo que pasa en pasos posteriores. Si tu algoritmo tiene un bucle, escríbelo como un bucle, y describe explícitamente lo que pasa en una iteración arbitraria. Similarmente, si tu algoritmo es recursivo, escríbelo recursivamente, y describe explícitamente los límites de los casos y lo que pasa en cada caso.

¹⁸ En particular, asumo que eres un novato escéptico!

¹⁹ Nunca he escuchado a nadie cantar "√2 Botellas de Cerveza en la Pared." Ocasionalmente he escuchado a teóricos de conjuntos cantando "ℵ₀ botellas de cerveza en la pared", pero por alguna razón siempre se rindieron antes de que la canción terminara.

²⁰ Esta es, por supuesto, una cuestión de convicción religiosa. Los lingüistas de sillón discuten incesantemente sobre la hipótesis Sapir-Whorf, que establece (más o menos) que las personas piensan sólo en las categorías impuestas por sus idiomas. Según una formulación extrema de este principio, algunos conceptos en un idioma simplemente no pueden ser entendidos por hablantes de otros idiomas, no sólo por avance tecnológico—¿Cómo traducirías "jump the shark" o "Fortnite streamer" al arameo?—sino por diferencias estructurales inherentes entre idiomas y culturas. Para una visión más escéptica, ver *The Language Instinct* de Steven Pinker. Admitidamente hay algo de fuerza a esta idea cuando se aplica a diferentes paradigmas de programación. (¿Qué es el combinador Y, otra vez? ¿Cómo funcionan las plantillas? ¿Qué es un Abstract Factory?) Afortunadamente, esas diferencias son demasiado sutiles para tener cualquier impacto en el material de este libro. Para un contraejemplo convincente, ver la monografía de Chris Okasaki *Functional Data Structures* y sus descendientes más recientes.

**0.6 Análisis de Algoritmos**

No es suficiente sólo escribir un algoritmo y decir "¡He aquí!" También debemos convencer a nuestra audiencia (¡y a nosotros mismos!) de que el algoritmo realmente hace lo que se supone que debe hacer, y que lo hace eficientemente.

**Corrección**

En algunos contextos de aplicación, es aceptable que los programas se comporten correctamente la mayoría del tiempo, en todas las entradas "razonables". No en este libro; requerimos algoritmos que siempre sean correctos, para todas las entradas posibles. Además, debemos demostrar que nuestros algoritmos son correctos; confiar en nuestros instintos, o probar algunos casos de prueba, no es suficiente. A veces la corrección es verdaderamente obvia, especialmente para algoritmos que has visto en cursos anteriores. Por otro lado, "obvio" es muy a menudo un sinónimo de "incorrecto". La mayoría de los algoritmos que discutimos en este curso requieren trabajo real para demostrar que son correctos. En particular, las demostraciones de corrección usualmente involucran inducción. Nos gusta la inducción. La inducción es nuestra amiga.²¹

Por supuesto, antes de que podamos demostrar formalmente que nuestro algoritmo hace lo que se supone que debe hacer, ¡tenemos que describir formalmente lo que se supone que debe hacer!

**Tiempo de Ejecución**

La manera más común de clasificar diferentes algoritmos para el mismo problema es por qué tan rápido ejecutan. Idealmente, queremos el algoritmo más rápido posible para cualquier problema particular. En muchos contextos de aplicación, es aceptable que los programas ejecuten eficientemente la mayoría del tiempo, en todas las entradas "razonables". No en este libro; requerimos algoritmos que siempre ejecuten eficientemente, incluso en el peor caso.

Pero ¿cómo medimos el tiempo de ejecución? Como ejemplo específico, ¿cuánto tiempo toma cantar la canción BotellaseDeCerveza(n)? Esto es obviamente una función del valor de entrada n, pero también depende de qué tan rápido puedas cantar. Algunos cantantes podrían tomar diez segundos para cantar un verso; otros podrían tomar veinte. La tecnología amplía las posibilidades aún más. Dictar la canción por telégrafo usando código Morse podría tomar un minuto completo por verso. Descargar un mp3 por la Web podría tomar una décima de segundo por verso. Duplicar el mp3 en la memoria principal de una computadora podría tomar sólo unos pocos microsegundos por verso.

Lo que es importante aquí es cómo el tiempo de canto cambia conforme n crece. Cantar BotellaseDeCerveza(2n) requiere aproximadamente el doble de tiempo que cantar BotellaseDeCerveza(n), sin importar qué tecnología se esté usando. Esto se refleja en el tiempo de canto asintótico Θ(n).

Podemos medir el tiempo contando cuántas veces el algoritmo ejecuta una cierta instrucción o alcanza un cierto hito en el "código". Por ejemplo, podríamos notar que la palabra "cerveza" se canta tres veces en cada verso de BotellaseDeCerveza, así que el número de veces que cantas "cerveza" es una buena indicación del tiempo total de canto. Para esta pregunta, podemos dar una respuesta exacta: BotellaseDeCerveza(n) menciona cerveza exactamente 3n + 3 veces.

Incidentalmente, hay muchas canciones con tiempo de canto cuadrático. Esta probablemente es familiar para la mayoría de hablantes de inglés:

NDaysOfChristmas(gifts[2 .. n]):

para i ← 1 hasta n

Cantar "En el día i de Navidad, mi amor verdadero me dio"

para j ← i hacia abajo hasta 2

Cantar " j gifts[j],"

si i > 1

Cantar "y"

Cantar "una perdiz en un peral."

La entrada a NDaysOfChristmas es una lista de n − 1 regalos, representada aquí como un arreglo. Es bastante fácil mostrar que el tiempo de canto es Θ(n²); en particular, el cantante menciona el nombre de un regalo ∑(i=1 a n) i = n(n + 1)/2 veces (contando la perdiz en el peral). También es fácil ver que durante los primeros n días de Navidad, mi amor verdadero me dio exactamente ∑(i=1 a n) ∑(j=1 a i) j = n(n + 1)(n + 2)/6 = Θ(n³) regalos.

Otras canciones de tiempo cuadrático incluyen "Old MacDonald Had a Farm", "There Was an Old Lady Who Swallowed a Fly", "Hole in the Bottom of the Sea", "Green Grow the Rushes O", "The Rattlin' Bog", "The Court Of King Caractacus", "The Barley-Mow", "If I Were Not Upon the Stage", "Star Trekkin'", "Ist das nicht ein Schnitzelbank?",²² "Il Pulcino Pio", "Minkurinn í hænsnakofanum", "Echad Mi Yodea", y "Το κοκοράκι". Para más ejemplos, consulta a tu preescolar favorito.

Alouette(lapart[1 .. n]):

Chantez « Alouette, gentille alouette, alouette, je te plumerai. »

pour tout i de 1 à n

Chantez « Je te plumerai lapart[i]. Je te plumerai lapart[i]. »

pour tout j de i à 1 ⟨⟨à rebours⟩⟩

Chantez « Et lapart[j] ! Et lapart[j] ! »

Chantez « Alouette! Alouette! Aaaaaa. . . »

Chantez « . . . alouette, gentille allouette, alouette, je te plumerai. »

Algunas canciones tienen tiempos de canto incluso más bizarros. Un ejemplo bastante moderno es "The TELNET Song" de Guy Steele, que realmente toma tiempo Θ(2ⁿ) para cantar los primeros n versos; Steele recomendó n = 4. Finalmente, hay algunas canciones que nunca terminan.²³

Excepto por "The TELNET Song", todas estas canciones se expresan más naturalmente como un pequeño conjunto de bucles anidados, así que sus tiempos de ejecución de canto pueden computarse usando sumatorias anidadas. El tiempo de ejecución de un algoritmo recursivo se expresa más fácilmente como una recurrencia. Por ejemplo, el algoritmo de multiplicación del campesino puede expresarse recursivamente como sigue:

x · y = { 0 si x = 0 { ⌊x/2⌋ · (y + y) si x es par { ⌊x/2⌋ · (y + y) + y si x es impar

Sea T(x, y) el número de operaciones de paridad, adición y mediación requeridas para calcular x · y. Esta función satisface la desigualdad recursiva T(x, y) ≤ T(⌊x/2⌋, 2y) + 2 con caso base T(0, y) = 0. Las técnicas descritas en el siguiente capítulo implican la cota superior T(x, y) = O(log x).

A veces el tiempo de ejecución de un algoritmo depende de una implementación particular de alguna estructura de datos subyacente o subrutina. Por ejemplo, el algoritmo de distribución Huntington-Hill ApportionCongress ejecuta en tiempo O(N + RI + (R − n)E), donde N denota el tiempo de ejecución de NewPriorityQueue, I denota el tiempo de ejecución de Insert, y E denota el tiempo de ejecución de ExtractMax. Bajo la suposición razonable de que R ≥ 2n (en promedio, cada estado obtiene al menos dos representantes), podemos simplificar esta cota a O(N + R(I + E)). El tiempo de ejecución preciso depende de la implementación de la cola de prioridad subyacente. El Bureau del Censo implementa la cola de prioridad como un arreglo no ordenado, lo que nos da N = I = Θ(1) y E = Θ(n), así que la implementación del Bureau del Censo de ApportionCongress ejecuta en tiempo O(Rn). Sin embargo, si implementamos la cola de prioridad como un heap binario o un arreglo ordenado como heap, tenemos N = Θ(1) e I = E = O(log n), así que el algoritmo total ejecuta en tiempo O(R log n).

Finalmente, a veces estamos interesados en recursos computacionales otros que el tiempo, como espacio, número de lanzamientos de moneda, número de fallas de caché o página, número de mensajes entre procesos, o el número de regalos que mi amor verdadero me dio. Estos recursos pueden analizarse usando las mismas técnicas usadas para analizar el tiempo de ejecución. Por ejemplo, la multiplicación de celosía de dos números de n dígitos requiere espacio O(n²) si escribimos todos los productos parciales antes de sumarlos, pero sólo espacio O(n) si los sumamos sobre la marcha.

²¹ Si la inducción no es tu amiga, tendrás dificultades con este libro.

²² ¡Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!

²³ Simplemente siguen y siguen, mi amigo.

**Ejercicios**

1. Describe y analiza un algoritmo eficiente que determine, dado un arreglo legal de piezas estándar en un tablero de ajedrez estándar, qué jugador ganará al ajedrez desde la posición inicial dada si ambos jugadores juegan perfectamente. [Pista: ¡Hay una solución trivial de una línea!]

♥1. (a) Identifica (o escribe) una canción que requiera tiempo Θ(n³) para cantar los primeros n versos.

(b) Identifica (o escribe) una canción que requiera tiempo Θ(n log n) para cantar los primeros n versos.

(c) Identifica (o escribe) una canción que requiera alguna otra cantidad extraña de tiempo para cantar los primeros n versos.

1. Los lectores cuidadosos podrían quejarse de que nuestro análisis de canciones como "n Botellas de Cerveza en la Pared" o "Los n Días de Navidad" es excesivamente simplista, porque números más grandes toman más tiempo en cantarse que números más cortos. Más generalmente, porque hay sólo tantas palabras de una longitud dada, conjuntos más grandes de palabras necesariamente contienen palabras más largas.²⁴ Podemos estimar más precisamente el tiempo de canto contando el número de sílabas cantadas, en lugar del número de palabras.

(a) ¿Cuánto tiempo toma cantar el entero n?

(b) ¿Cuánto tiempo toma cantar "n Botellas de Cerveza en la Pared"?

(c) ¿Cuánto tiempo toma cantar "Los n Días de Navidad"?

Como siempre, expresa tus respuestas en la forma O(f(n)) para alguna función f.

1. La canción bebida acumulativa "The Barley Mow" ha sido cantada por todas las Islas Británicas durante siglos. La canción tiene muchas variantes; la Figura 0.6 contiene pseudoletra para una versión tradicionalmente cantada en Devon y Cornwall, donde vessel[i] es el nombre de un recipiente que contiene 2ⁱ onzas de cerveza.²⁵

BarleyMow(n):

"Here's a health to the barley-mow, my brave boys,"

"Here's a health to the barley-mow!"

"We'll drink it out of the jolly brown bowl,"

"Here's a health to the barley-mow!"

"Here's a health to the barley-mow, my brave boys,"

"Here's a health to the barley-mow!"

para i ← 1 hasta n

"We'll drink it out of the vessel[i], boys,"

"Here's a health to the barley-mow!"

para j ← i hacia abajo hasta 1

"The vessel[j],"

"And the jolly brown bowl!"

"Here's a health to the barley-mow!"

"Here's a health to the barley-mow, my brave boys,"

"Here's a health to the barley-mow!"

**Figura 0.6.** "The Barley Mow".

(a) Supón que cada nombre vessel[i] es una sola palabra, y puedes cantar cuatro palabras por segundo. ¿Cuánto tiempo te tomaría cantar BarleyMow(n)? (Da una cota asintótica ajustada.)

(b) Si quieres cantar esta canción para valores arbitrariamente grandes de n, tendrás que inventar tus propios nombres de recipientes. Para evitar repetición, estos nombres deben volverse progresivamente más largos conforme n aumenta. Supón que vessel[n] tiene Θ(log n) sílabas, y puedes cantar seis sílabas por segundo. Ahora ¿cuánto tiempo te tomaría cantar BarleyMow(n)? (Da una cota asintótica ajustada.)

(c) Supón que cada vez que mencionas el nombre de un recipiente, realmente bebes la cantidad correspondiente de cerveza: una onza para el jolly brown bowl, y 2ⁱ onzas para cada vessel[i]. Asumiendo para propósitos de este problema que tienes al menos 21 años de edad, ¿exactamente cuántas onzas de cerveza beberías si cantaras BarleyMow(n)? (Da una respuesta exacta, no sólo una cota asintótica.)

1. Recuerda que la entrada al algoritmo Huntington-Hill ApportionCongress es un arreglo Pop[1 .. n], donde Pop[i] es la población del i-ésimo estado, y un entero R, el número total de representantes a ser asignados. La salida es un arreglo Rep[1 .. n], donde Rep[i] es el número de representantes asignados al i-ésimo estado por el algoritmo.

El algoritmo Huntington-Hill a veces se describe de una manera que evita el uso de colas de prioridad enteramente. El algoritmo de nivel superior "adivina" un número real positivo D, llamado el divisor, y luego ejecuta la siguiente subrutina para calcular una distribución. La variable q es la cuota ideal de representantes asignados a un estado para el divisor dado D; el número real de representantes asignados es siempre ya sea ⌈q⌉ o ⌊q⌋.

HHGuess(Pop[1 .. n],R, D):

reps ← 0

para i ← 1 hasta n

q ← Pop[i]/D

si q · q < ⌈q⌉ · ⌊q⌋

Rep[i] ← ⌊q⌋

sino

Rep[i] ← ⌈q⌉

reps ← reps + Rep[i]

retornar reps

Hay tres posibilidades para el valor de retorno final reps. Si reps < R, no asignamos suficientes representantes, lo que (al menos intuitivamente) significa que nuestro divisor D era demasiado pequeño. Si reps > R, asignamos demasiados representantes, lo que (al menos intuitivamente) significa que nuestro divisor D era demasiado grande. Finalmente, si reps = R, podemos retornar el arreglo Rep[1 .. n] como la distribución final. En la práctica, podemos calcular una distribución válida (con reps = R) llamando HHGuess con un pequeño número de divisores enteros cerca del divisor estándar D = P/R.

En los siguientes problemas, sea P = ∑(i=1 a n) Pop[i] la población total de todos los n estados, y asume que n ≤ R ≤ P.

²⁴ ¡Ja, das ist das Subatomarteilchenbeschleunigungsnaturmäßigkeitsuntersuchungsmaschine!

²⁵ En la práctica, la canción usa algún subconjunto de los siguientes recipientes: nipperkin, quarter-gill, half-a-gill, gill, quarter-pint, half-a-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel/kilderkin, barrel, hogshead, pipe/butt, tun, well, river, y ocean. Con pocas excepciones (especialmente al final), cada recipiente en esta lista tiene el doble del volumen de su predecesor. Las versiones irlandesas y escocesas de la canción tienen letras ligeramente diferentes, y usualmente cambian a personas (barmaid, landlord, drayer, etc.) después de "gallon".

Una versión temprana de la canción titulada "Give us once a drink" aparece en la obra *Jack Drum's Entertainment* (o la *Comedie of Pasquill and Katherine*) escrita por John Marston alrededor de 1600. ("Giue vs once a drinke for and the black bole. Sing gentle Butler bally moy!") Hay algún desacuerdo sobre si Marston escribió la "high Dutch Song" específicamente para la obra, si "bally moy" es una mondegrin para "barley mow" o viceversa, o si es realmente la misma canción en absoluto. Estas discusiones se tienen mejor sobre n botellas de cerveza.

**TRADUCCIÓN - ALGORITMOS: RECURSIÓN**

**0. INTRODUCCIÓN**

(a) Demuestra que llamar a HHGuess con el divisor estándar D = P/R no necesariamente produce una distribución válida.

(b) Demuestra que si HHGuess devuelve el mismo valor de representantes para dos divisores diferentes D y D', también calcula la misma asignación Rep[1 .. n] para ambos divisores.

(c) Demuestra que si HHGuess devuelve el valor correcto R, calcula la misma asignación Rep[1 .. n] que nuestro algoritmo anterior ApportionCongress.

(d) ¡Demuestra que un divisor "correcto" D no necesariamente existe! Es decir, describe entradas Pop[1 .. n] y R, donde n ≤ R ≤ P, tal que para cada número real D > 0, el número de representantes asignados por HHGuess no es igual a R. [Pista: ¿Qué pasa si cambiamos < por ≤ en la cuarta línea de HHGuess?]

*El control de una fuerza grande es el mismo principio que el control de unos pocos hombres: es simplemente una cuestión de dividir sus números.* — Sun Zi, El Arte de la Guerra (c. 400CE), traducido por Lionel Giles (1910)

*Nuestra vida se desvanece por el detalle... Simplifica, simplifica.* — Henry David Thoreau, Walden (1854)

*Ahora, no me preguntes qué es Voom. Nunca lo sabré. ¡Pero, muchacho! Déjame decirte, ¡SÍ limpia la nieve!* — Dr. Seuss [Theodor Seuss Geisel], El Gato en el Sombrero Regresa (1958)

*Haz primero los trabajos difíciles. Los trabajos fáciles se cuidarán solos.* — atribuido a Dale Carnegie

**1**

**Recursión**

**1.1 Reducciones**

La reducción es la técnica más común utilizada en el diseño de algoritmos. Reducir un problema X a otro problema Y significa escribir un algoritmo para X que use un algoritmo para Y como una caja negra o subrutina. Crucialmente, la corrección del algoritmo resultante para X no puede depender de ninguna manera en cómo funciona el algoritmo para Y. Lo único que podemos asumir es que la caja negra resuelve Y correctamente. El funcionamiento interno de la caja negra simplemente no es asunto nuestro; es el problema de alguien más. A menudo es mejor pensar literalmente en la caja negra como funcionando puramente por magia.

Por ejemplo, el algoritmo de multiplicación campesina descrito en el capítulo anterior reduce el problema de multiplicar dos enteros positivos arbitrarios a tres problemas más simples: suma, mediación (dividir por la mitad) y verificación de paridad. El algoritmo se basa en un tipo de datos abstracto "entero positivo" que soporta esas tres operaciones, pero la corrección del algoritmo de multiplicación no depende de la representación precisa de los datos (marcas de cuenta, fichas de arcilla, hexagesimal babilónico, quipu, varillas de contar, numerales romanos, posiciones de dedos, piedras de augrym, numerales gobar, binario, negabinario, código Gray, ternario balanceado, finary, cuater-imaginario, ...), o en las implementaciones precisas de esas operaciones.

Por supuesto, el tiempo de ejecución del algoritmo de multiplicación depende del tiempo de ejecución de las operaciones de suma, mediación y paridad, pero eso es un tema separado de la corrección. Más importante aún, podemos crear un algoritmo de multiplicación más eficiente simplemente cambiando a una representación numérica más eficiente (de marcas de cuenta a notación posicional, por ejemplo).

De manera similar, el algoritmo de Huntington-Hill reduce el problema de distribución del Congreso al problema de mantener una cola de prioridad que soporta las operaciones Insert y ExtractMax. El tipo de datos abstracto "cola de prioridad" es una caja negra; la corrección del algoritmo de distribución no depende de ninguna estructura de datos específica de cola de prioridad. Por supuesto, el tiempo de ejecución del algoritmo de distribución depende del tiempo de ejecución de los algoritmos Insert y ExtractMax, pero eso es un tema separado de la corrección del algoritmo. La belleza de la reducción es que podemos crear un algoritmo de distribución más eficiente simplemente intercambiando una nueva estructura de datos de cola de prioridad. Además, el diseñador de esa estructura de datos no necesita saber o preocuparse de que será utilizada para distribuir el Congreso.

Cuando diseñamos algoritmos, puede que no sepamos exactamente cómo están implementados los bloques de construcción básicos que usamos, o cómo nuestros algoritmos podrían ser utilizados como bloques de construcción para resolver problemas aún más grandes. Esa ignorancia es incómoda para muchos principiantes, pero es tanto inevitable como extremadamente útil. Incluso cuando sí sabes precisamente cómo funcionan tus componentes, a menudo es extremadamente útil pretender que no lo sabes.

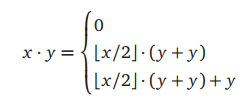
**1.2 Simplifica y Delega**

La recursión es un tipo particularmente poderoso de reducción, que puede describirse vagamente como sigue:

• Si la instancia dada del problema puede resolverse directamente, resuélvela directamente. • De lo contrario, redúcela a una o más instancias más simples del mismo problema.

Si la autorreferencia es confusa, puede ser útil imaginar que alguien más va a resolver los problemas más simples, tal como asumirías para otros tipos de reducciones. Me gusta llamar a ese alguien más el Hada de la Recursión. Tu única tarea es simplificar el problema original, o resolverlo directamente cuando la simplificación es innecesaria o imposible; el Hada de la Recursión resolverá todos los subproblemas más simples para ti, usando Métodos Que No Son Asunto Tuyo Así Que No Te Metas¹. Los lectores matemáticamente sofisticados podrían reconocer al Hada de la Recursión por su nombre más formal: la Hipótesis de Inducción.

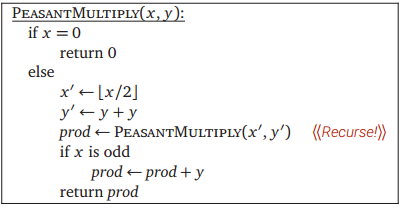
Hay una condición técnica leve que debe satisfacerse para que cualquier método recursivo funcione correctamente: No debe haber una secuencia infinita de reducciones a instancias más simples y más simples. Eventualmente, las reducciones recursivas deben llevar a un caso base elemental que pueda resolverse por algún otro método; de lo contrario, el algoritmo recursivo se ejecutará para siempre. La forma más común de satisfacer esta condición es reducir a una o más instancias más pequeñas del mismo problema. Por ejemplo, si la entrada original es un skreeble con n glurps, la entrada a cada llamada recursiva debería ser un skreeble con estrictamente menos de n glurps. Por supuesto, esto es imposible si el skreeble no tiene glurps en absoluto —¡No puedes tener glurps negativos; eso sería tonto!— así que en ese caso debemos grindlebloff el skreeble usando algún otro método.

Ya hemos visto una instancia de este patrón en el algoritmo de multiplicación campesina, que se basa directamente en la siguiente identidad recursiva.

If x =0

If x es par

If x es impar

La misma recurrencia puede expresarse algorítmicamente como sigue:

Un escriba egipcio perezoso podría ejecutar este algoritmo calculando x' e y', pidiendo a un escriba más junior que multiplique x' e y', y luego posiblemente agregando y a la respuesta del escriba junior. El problema del escriba junior es más simple porque x' < x, y disminuir repetidamente un entero positivo eventualmente lleva a 0. Cómo el escriba junior realmente calcula x' · y' no es asunto del escriba senior (y tampoco es asunto tuyo).

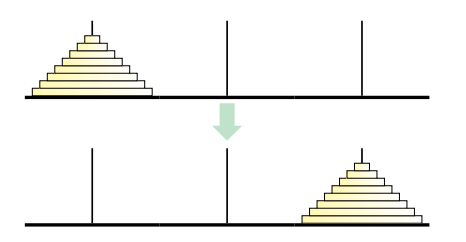
------------------------------------------------------------

¹Cuando era estudiante universitario, atribuía la recursión a "duendes" en lugar del Hada de la Recursión, refiriéndome al cuento de los Hermanos Grimm sobre un viejo zapatero que deja su trabajo sin terminar cuando se va a dormir, solo para descubrir al despertar que los duendes ("Wichtelmänner") han terminado todo durante la noche. Alguien más experimentado enteogénicamente que yo podría reconocer a estos Rekursionswichtelmänner como los "duendes máquina auto-transformadores" de Terence McKenna.

**1.3 Torre de Hanoi**

El rompecabezas de la Torre de Hanoi fue publicado por primera vez —¡como un rompecabezas físico real!— por el maestro y matemático recreacional francés Édouard Lucas en 1883², bajo el pseudónimo "N. Claus (de Siam)" (un anagrama de "Lucas d'Amiens"). Al año siguiente, Henri de Parville describió el rompecabezas con la siguiente historia notable³:

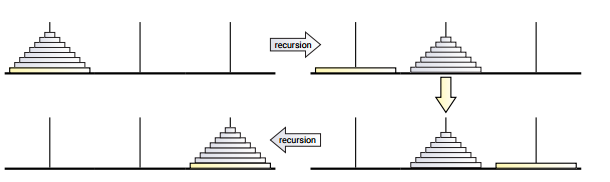
En el gran templo de Benares⁴... bajo la cúpula que marca el centro del mundo, descansa una placa de latón en la cual están fijadas tres agujas de diamante, cada una de un codo de altura y tan gruesa como el cuerpo de una abeja. En una de estas agujas, en la creación, Dios colocó sesenta y cuatro discos de oro puro, el disco más grande descansando en la placa de latón, y los otros volviéndose más pequeños y más pequeños hasta el de arriba. Esta es la Torre de Bramah. Día y noche sin cesar los sacerdotes transfieren los discos de una aguja de diamante a otra según las leyes fijas e inmutables de Bramah, que requieren que el sacerdote de turno no debe mover más de un disco a la vez y que debe colocar este disco en una aguja para que no haya un disco más pequeño debajo de él. Cuando los sesenta y cuatro discos hayan sido así transferidos desde la aguja en la cual en la creación Dios los colocó a una de las otras agujas, torre, templo y brahmanes por igual se desmoronarán en polvo, y con un trueno el mundo se desvanecerá.



**Figura 1.1.** El rompecabezas de la Torre de Hanoi (8 discos)

Por supuesto, como buenos científicos de la computación, nuestro primer instinto al leer esta historia es sustituir la variable n por la constante codificada 64. Y porque la mayoría de las instancias físicas del rompecabezas están hechas de madera en lugar de diamantes y oro, llamaré a las tres ubicaciones posibles para los discos "clavijas" en lugar de "agujas". ¿Cómo podemos mover una torre de n discos de una clavija a otra, usando una tercera clavija libre como marcador de posición ocasional, sin nunca colocar un disco encima de un disco más pequeño?

Como N. Claus (de Siam) señaló en el panfleto incluido con su rompecabezas, el secreto para resolver este rompecabezas es pensar recursivamente. En lugar de tratar de resolver todo el rompecabezas de una vez, concentrémonos en mover solo el disco más grande. No podemos moverlo al principio, porque todos los otros discos están en el camino. Así que primero tenemos que mover esos n − 1 discos más pequeños a la clavija libre. Una vez que eso esté hecho, podemos mover el disco más grande directamente a su destino. Finalmente, para terminar el rompecabezas, tenemos que mover los n − 1 discos más pequeños desde la clavija libre a su destino.



**Figura 1.2.** El algoritmo de la Torre de Hanoi; ignora todo excepto el disco inferior.

Así que ahora todo lo que tenemos que averiguar es cómo—

¡¡NO!! ¡¡ALTO!!

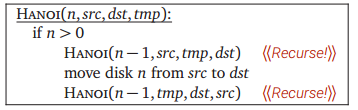
¡Eso es todo! ¡Hemos terminado! Hemos reducido exitosamente el problema de la Torre de Hanoi de n discos a dos instancias del problema de la Torre de Hanoi de (n − 1) discos, que podemos entregar alegremente al Hada de la Recursión —o para continuar con la metáfora de Lucas, a los monjes junior del templo. Nuestro trabajo ha terminado. Si no confiáramos en los monjes junior, no los habríamos contratado; déjalos hacer su trabajo en paz.

Nuestra reducción hace una suposición sutil pero extremadamente importante: Hay un disco más grande. Nuestro algoritmo recursivo funciona para cualquier número positivo de discos, pero se descompone cuando n = 0. Debemos manejar ese caso usando un método diferente. Afortunadamente, los monjes en Benares, siendo buenos budistas, son bastante hábiles en mover cero discos de una clavija a otra en nada de tiempo, no haciendo nada.



**Figura 1.3.** El caso base vacuo para el algoritmo de la Torre de Hanoi. No hay cuchara.

Puede ser tentador pensar en cómo se mueven todos esos discos más pequeños —o más generalmente, qué sucede cuando se desenrolla la recursión— pero realmente, no lo hagas. Para la mayoría de los algoritmos recursivos, desenrollar la recursión no es necesario ni útil. Nuestra única tarea es reducir la instancia del problema que nos dan a una o más instancias más simples, o resolver el problema directamente si tal reducción es imposible. Nuestro algoritmo recursivo de la Torre de Hanoi es trivialmente correcto cuando n = 0. Para cualquier n ≥ 1, el Hada de la Recursión mueve correctamente los n − 1 discos superiores (más formalmente, la Hipótesis de Inducción implica que nuestro algoritmo recursivo mueve correctamente los n − 1 discos superiores) así que nuestro algoritmo es correcto.

El algoritmo recursivo de Hanoi se expresa en pseudocódigo en la Figura 1.4. El algoritmo mueve una pila de n discos desde una clavija fuente (src) a una clavija destino (dst) usando una tercera clavija temporal (tmp) como marcador de posición. Nota que el algoritmo correctamente no hace nada en absoluto cuando n = 0.

**Figura 1.4.** Un algoritmo recursivo para resolver la Torre de Hanoi

Sea T(n) el número de movimientos requeridos para transferir n discos —el tiempo de ejecución de nuestro algoritmo. Nuestro caso base vacuo implica que T(0) = 0, y el algoritmo recursivo más general implica que T(n) = 2T(n − 1) + 1 para cualquier n ≥ 1. Escribiendo los primeros varios valores de T(n), podemos fácilmente suponer que T(n) = 2ⁿ − 1; una prueba de inducción directa implica que esta suposición es correcta. En particular, mover una torre de 64 discos requiere 2⁶⁴ − 1 = 18,446,744,073,709,551,615 movimientos individuales. Así, incluso a la impresionante velocidad de un movimiento por segundo, los monjes en Benares estarán trabajando durante aproximadamente 585 mil millones de años ("plus de cinq milliards de siècles") antes de que torre, templo y brahmanes por igual se desmoronen en polvo, y con un trueno el mundo se desvanezca.

**1.4 Mergesort**

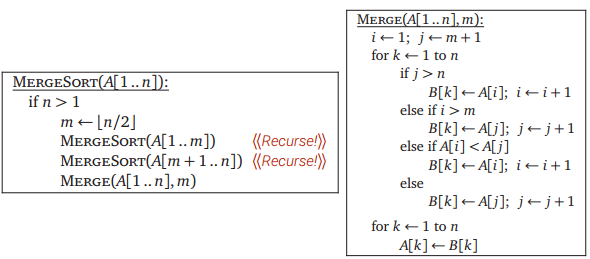
Mergesort es uno de los primeros algoritmos diseñados para computadoras de programa almacenado de propósito general. El algoritmo fue desarrollado por John von Neumann en 1945, y descrito en detalle en una publicación con Herman Goldstine en 1947, como uno de los primeros programas no numéricos para la EDVAC⁵.

1. Divide el arreglo de entrada en dos subarreglos de tamaño aproximadamente igual.
2. Ordena recursivamente cada uno de los subarreglos usando mergesort.
3. Combina los subarreglos recién ordenados en un solo arreglo ordenado.

**Entrada:** S O R T I N G E X A M P L E  
**Dividir:** S O R T I N G | E X A M P L E  
**Recurrir Izquierda:** I N O R S T G | E X A M P L E  
**Recurrir Derecha:** I N O R S T G | A E G L M P X  
**Combinar:** A E G I L M N O P R S T X

**Figura 1.5.** Un ejemplo de mergesort.

El primer paso es completamente trivial —solo dividir el tamaño del arreglo por dos— y podemos delegar el segundo paso al Hada de la Recursión. Todo el trabajo real se hace en el paso final de combinación. Una descripción completa del algoritmo se da en la Figura 1.6; para mantener clara la estructura recursiva, he extraído el paso de combinación en una subrutina independiente. El algoritmo de combinación también es recursivo —identifica el primer elemento del arreglo de salida, y luego combina recursivamente el resto de los arreglos de entrada.



**Figura 1.6.** Mergesort

**Corrección**

Para probar que este algoritmo es correcto, aplicamos nuestro viejo amigo inducción dos veces, primero a la subrutina Merge luego al algoritmo MergeSort de nivel superior.

**Lema 1.1.** Merge combina correctamente los subarreglos A[1 .. m] y A[m + 1 .. n], asumiendo que esos subarreglos están ordenados en la entrada.

**Prueba:** Sea A[1 .. n] cualquier arreglo y m cualquier entero tal que los subarreglos A[1 .. m] y A[m+1 .. n] estén ordenados. Demostramos que para todo k de 0 a n, las últimas n − k − 1 iteraciones del bucle principal combinan correctamente A[i .. m] y A[j .. n] en B[k .. n]. La prueba procede por inducción en n − k + 1, el número de elementos que quedan por combinar.

Si k > n, el algoritmo combina correctamente los dos subarreglos vacíos no haciendo absolutamente nada. (Este es el caso base de la prueba inductiva.) De lo contrario, hay cuatro casos a considerar para la k-ésima iteración del bucle principal.

• Si j > n, entonces el subarreglo A[j .. n] está vacío, así que min{A[i .. m] ∪ A[j .. n]} = A[i].

• Si i > m, entonces el subarreglo A[i .. m] está vacío, así que min{A[i .. m] ∪ A[j .. n]} = A[j].

• De lo contrario, si A[i] < A[j], entonces min{A[i .. m] ∪ A[j .. n]} = A[i].

• De lo contrario, debemos tener A[i] ≥ A[j], y min{A[i .. m] ∪ A[j .. n]} = A[j].

En los cuatro casos, B[k] se asigna correctamente al elemento más pequeño de A[i .. m] ∪ A[j .. n]. En los dos casos con la asignación B[k] ← A[i], el Hada de la Recursión combina correctamente —perdón, quiero decir que la Hipótesis de Inducción implica que las últimas n − k iteraciones del bucle principal combinan correctamente A[i + 1 .. m] y A[j .. n] en B[k + 1 .. n]. De manera similar, en los otros dos casos, el Hada de la Recursión también combina correctamente el resto de los subarreglos.

**Teorema 1.2.** MergeSort ordena correctamente cualquier arreglo de entrada A[1 .. n].

**Prueba:** Demostramos el teorema por inducción en n. Si n ≤ 1, el algoritmo correctamente no hace nada. De lo contrario, el Hada de la Recursión ordena correctamente —perdón, quiero decir que la hipótesis de inducción implica que nuestro algoritmo ordena correctamente los dos subarreglos más pequeños A[1 .. m] y A[m + 1 .. n], después de lo cual se combinan correctamente en un solo arreglo ordenado (por el Lema 1.1). □

**Análisis**

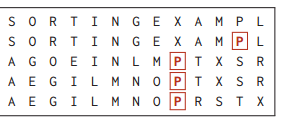
Porque el algoritmo MergeSort es recursivo, su tiempo de ejecución se expresa naturalmente como una recurrencia. Merge claramente toma tiempo O(n), porque es un bucle for simple con trabajo constante por iteración. Obtenemos inmediatamente la siguiente recurrencia para MergeSort:

T(n) = T(⌈n/2⌉) + T(⌊n/2⌋) + O(n).

Como en la mayoría de las recurrencias de divide y vencerás, podemos descartar con seguridad los pisos y techos (usando una técnica llamada transformaciones de dominio descrita más adelante en este capítulo), dándonos la recurrencia más simple T(n) = 2T(n/2) + O(n). El caso "todos los niveles iguales" del método del árbol de recursión (también descrito más adelante en este capítulo) implica inmediatamente la solución de forma cerrada T(n) = O(n log n). Incluso si no estás (aún) familiarizado con árboles de recursión, puedes verificar la solución T(n) = O(n log n) por inducción.

**1.5 Quicksort**

Quicksort es otro algoritmo de ordenamiento recursivo, descubierto por Tony Hoare en 1959 y publicado por primera vez en 1961. En este algoritmo, el trabajo duro es dividir el arreglo en subarreglos más pequeños antes de la recursión, para que combinar los subarreglos ordenados sea trivial.

1. Elige un elemento pivote del arreglo.
2. Particiona el arreglo en tres subarreglos que contengan los elementos menores que el pivote, el elemento pivote mismo, y los elementos mayores que el pivote.
3. ****Ordena recursivamente el primer y último subarreglos usando quicksort.

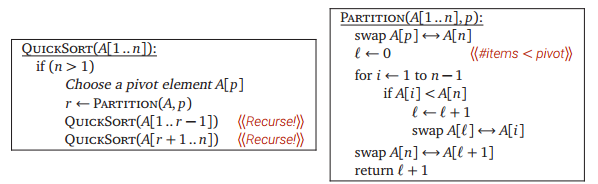
**Entrada:   
Elegir un pivote:**

**Particionar:**

**Recurrir Izquierda:   
Recurrir Derecha:**

**Figura 1.7.** Un ejemplo de quicksort.

Se da pseudocódigo más detallado en la Figura 1.8. En la subrutina Partition, el parámetro de entrada p es el índice del elemento pivote en el arreglo no ordenado; la subrutina particiona el arreglo y devuelve el nuevo índice del elemento pivote. Hay muchos algoritmos de partición eficientes diferentes; el que estoy presentando aquí se atribuye a Nico Lomuto⁶. La variable ℓ cuenta el número de elementos en el arreglo que son menores que el elemento pivote.



**Figura 1.8.** Quicksort

**Corrección**

Al igual que mergesort, probar que QuickSort es correcto requiere dos pruebas de inducción separadas: una para probar que Partition particiona correctamente el arreglo, y la otra para probar que QuickSort ordena correctamente asumiendo que Partition es correcto. Para probar que Partition es correcto, necesitamos probar el siguiente invariante del bucle: Al final de cada iteración del bucle principal, todo en el subarreglo A[1 .. ℓ] es menor que A[n], y nada en el subarreglo A[ℓ + 1 .. i] es menor que A[n]. Dejaré los detalles directos pero tediosos restantes como ejercicios para el lector.

**Análisis**

El análisis de quicksort también es similar al de mergesort. Partition claramente se ejecuta en tiempo O(n), porque es un bucle for simple con trabajo constante por iteración. Para QuickSort, obtenemos una recurrencia que depende de r, el rango del elemento pivote elegido:

T(n) = T(r − 1) + T(n − r) + O(n)

Si pudiéramos de alguna manera elegir mágicamente siempre el pivote para que sea el elemento mediano del arreglo A, tendríamos r = ⌈n/2⌉, los dos subproblemas serían lo más cercano posible al mismo tamaño, la recurrencia se convertiría en

T(n) = T(⌈n/2⌉ − 1) + T(⌊n/2⌋) + O(n) ≤ 2T(n/2) + O(n),

y tendríamos T(n) = O(n log n) usando el método del árbol de recursión o el método aún más simple "Oh sí, ya resolvimos esa recurrencia para mergesort".

De hecho, como veremos más adelante en este capítulo, realmente podemos localizar el elemento mediano en un arreglo no ordenado en tiempo lineal, pero el algoritmo es bastante complicado, y la constante oculta en la notación O(·) es lo suficientemente grande como para hacer que el algoritmo de ordenamiento resultante sea impráctico. En la práctica, la mayoría de los programadores se conforman con algo simple, como elegir el primer o último elemento del arreglo. En este caso, r puede tomar cualquier valor entre 1 y n, así que tenemos

T(n) = max₁≤r≤n {T(r − 1) + T(n − r) + O(n)}.

En el peor caso, los dos subproblemas están completamente desequilibrados —ya sea r = 1 o r = n— y la recurrencia se convierte en T(n) ≤ T(n − 1) + O(n). La solución es T(n) = O(n²).

Otra heurística común se llama "mediana de tres" —elegir tres elementos (usualmente al principio, en el medio y al final del arreglo), y tomar la mediana de esos tres elementos como el pivote. Aunque esta heurística es algo más eficiente en la práctica que solo elegir un elemento, especialmente cuando el arreglo ya está (casi) ordenado, todavía podemos tener r = 2 o r = n − 1 en el peor caso. Con la heurística mediana de tres, la recurrencia se convierte en T(n) ≤ T(1) + T(n − 2) + O(n), cuya solución sigue siendo T(n) = O(n²).

Intuitivamente, el elemento pivote debería "usualmente" caer en algún lugar en el medio del arreglo, digamos con rango entre n/10 y 9n/10. Esta observación sugiere que el tiempo de ejecución de "caso promedio" debería ser O(n log n). Aunque esta intuición puede formalizarse, la formalización más común hace la suposición completamente irrealista de que todas las permutaciones del arreglo de entrada son igualmente probables. ¡Los datos del mundo real pueden ser aleatorios, pero no son aleatorios de ninguna manera que podamos predecir de antemano, y ciertamente no son uniformes!⁷

Ocasionalmente la gente también considera el tiempo de ejecución de "mejor caso" por alguna razón. No lo haremos.

**1.6 El Patrón**

Tanto mergesort como quicksort siguen un patrón general de tres pasos llamado divide y vencerás:

1. Divide la instancia dada del problema en varias instancias independientes más pequeñas de exactamente el mismo problema.
2. Delega cada instancia más pequeña al Hada de la Recursión.
3. Combina las soluciones para las instancias más pequeñas en la solución final para la instancia dada.

Si el tamaño de cualquier instancia cae por debajo de algún umbral constante, abandonamos la recursión y resolvemos el problema directamente, por fuerza bruta, en tiempo constante.

Probar que un algoritmo de divide y vencerás es correcto casi siempre requiere inducción. Analizar el tiempo de ejecución requiere establecer y resolver una recurrencia, que usualmente (¡pero desafortunadamente no siempre!) puede resolverse usando árboles de recursión.

**1.7 Árboles de Recursión**

Entonces, ¿qué son estos "árboles de recursión" de los que sigo hablando? Los árboles de recursión son una herramienta simple, general y pictórica para resolver recurrencias de divide y vencerás. Un árbol de recursión es un árbol con raíz con un nodo para cada subproblema recursivo. El valor de cada nodo es la cantidad de tiempo gastado en el subproblema correspondiente excluyendo las llamadas recursivas. Así, el tiempo total de ejecución del algoritmo es la suma de los valores de todos los nodos en el árbol.

Para hacer esta idea más concreta, imagina un algoritmo de divide y vencerás que gasta O(f(n)) tiempo en trabajo no recursivo, y luego hace r llamadas recursivas, cada una en un problema de tamaño n/c. Hasta factores constantes (que podemos ocultar en la notación O()), el tiempo de ejecución de este algoritmo está gobernado por la recurrencia

T(n) = r T(n/c) + f(n).

La raíz del árbol de recursión para T(n) tiene valor f(n) y r hijos, cada uno de los cuales es la raíz de un árbol de recursión (definido recursivamente) para T(n/c). Equivalentemente, un árbol de recursión es un árbol r-ario completo donde cada nodo en profundidad d contiene el valor f(n/c^d). (Siéntete libre de asumir que n es una potencia entera de c, para que n/c^d sea siempre un entero, aunque de hecho esto no importa.)

En la práctica, recomiendo dibujar los primeros dos o tres niveles del árbol, como en la Figura 1.9.

**Figura 1.9.** Un árbol de recursión para la recurrencia T(n) = r T(n/c) + f(n)

Las hojas del árbol de recursión corresponden a los casos base de la recurrencia. Porque solo estamos buscando límites asintóticos, el caso base preciso en realidad no importa; podemos asumir con seguridad T(n) = 1 para todo n ≤ n₀, donde n₀ es una constante positiva arbitraria. En particular, podemos elegir cualquier valor de n₀ que sea más conveniente para nuestro análisis. Para este ejemplo, elegiré n₀ = 1.

Ahora T(n) es la suma de todos los valores en el árbol de recursión; podemos evaluar esta suma considerando el árbol nivel por nivel. Para cada entero i, el i-ésimo nivel del árbol tiene exactamente r^i nodos, cada uno con valor f(n/c^i). Así,

T(n) = Σᵢ₌₀ᴸ r^i · f(n/c^i) (Σ)

donde L es la profundidad del árbol. Nuestro caso base n₀ = 1 implica inmediatamente L = log\_c n, porque n/c^L = n₀ = 1. Se sigue que el número de hojas en el árbol de recursión es exactamente r^L = r^(log\_c n) = n^(log\_c r). Así, el último término en la suma nivel por nivel (Σ) es n^(log\_c r) · f(1) = O(n^(log\_c r)), porque f(1) = O(1).

Hay tres casos comunes donde la serie nivel por nivel (Σ) es especialmente fácil de evaluar:

• **Decreciente:** Si la serie decae exponencialmente —cada término es un factor constante más pequeño que el término anterior— entonces T(n) = O(f(n)). En este caso, la suma está dominada por el valor en la raíz del árbol de recursión.

• **Igual:** Si todos los términos en la serie son iguales, inmediatamente tenemos T(n) = O(f(n) · L) = O(f(n) log n). (La constante c se desvanece en la notación O().)

• **Creciente:** Si la serie crece exponencialmente —cada término es un factor constante más grande que el término anterior— entonces T(n) = O(n^(log\_c r)). En este caso, la suma está dominada por el número de hojas en el árbol de recursión.

En el primer y tercer casos, solo el término más grande en la serie geométrica importa; todos los otros términos son absorbidos por la notación O(·). En el caso decreciente, ni siquiera tenemos que calcular L; el límite superior asintótico seguiría siendo válido si el árbol de recursión fuera infinito!

Como ejemplo elemental, si dibujamos los primeros niveles del árbol de recursión para la recurrencia (simplificada) de mergesort T(n) = 2T(n/2) + O(n), descubrimos que todos los niveles son iguales, lo que implica inmediatamente T(n) = O(n log n).

**Figura 1.10.** El árbol de recursión para mergesort

La técnica del árbol de recursión también puede usarse para algoritmos donde los subproblemas recursivos tienen tamaños diferentes. Por ejemplo, si pudiéramos de alguna manera implementar quicksort para que el pivote siempre caiga en el tercio medio del arreglo ordenado, el tiempo de ejecución en el peor caso satisfaría la recurrencia

T(n) ≤ T(n/3) + T(2n/3) + O(n).

Esta recurrencia podría verse aterradora, pero en realidad es bastante manejable. Si dibujamos algunos niveles del árbol de recursión resultante, rápidamente nos damos cuenta de que la suma de valores en cualquier nivel es a lo sumo n —los niveles más profundos podrían estar perdiendo algunos nodos— y todo el árbol tiene profundidad log₃/₂ n = O(log n). Se sigue inmediatamente que T(n) = O(n log n). (Además, el número de niveles completos en el árbol de recursión es log₃ n = Ω(log n), así que este análisis conservador puede mejorarse por a lo sumo un factor constante, lo que para nuestros propósitos significa que no puede mejorarse en absoluto.) El hecho de que el árbol de recursión esté desequilibrado simplemente no importa.

Como ejemplo más extremo, la recurrencia de peor caso para quicksort T(n) = T(n − 1) + T(1) + O(n) nos da un árbol de recursión completamente desequilibrado, donde un hijo de cada nodo interno es una hoja. La suma nivel por nivel no cae en ninguna de nuestras tres categorías por defecto, pero aún podemos derivar la solución T(n) = O(n²) observando que cada valor de nivel es a lo sumo n y hay a lo sumo n niveles. (Nuevamente, este análisis conservador es ajustado, porque n/2 niveles cada uno tiene valor al menos n/2.)

**Figura 1.11.** Árboles de recursión para quicksort con buenos pivotes (izquierda) y con pivotes de peor caso (derecha)

**♥ Ignorar Pisos y Techos Está Bien, En Serio**

Los lectores cuidadosos podrían objetar que nuestro análisis pasa por alto un detalle importante. El tiempo de ejecución de mergesort no obedece realmente la recurrencia T(n) = 2T(n/2) + O(n); después de todo, el tamaño de entrada n podría ser impar, ¿y qué podría significar posiblemente ordenar un arreglo de tamaño 421/2 o 177/8? La recurrencia real de mergesort es algo más desordenada:

T(n) = T(⌈n/2⌉) + T(⌊n/2⌋) + O(n).

Claro, podríamos verificar que T(n) = O(n log n) usando inducción, pero los cálculos necesarios serían horribles. Afortunadamente, hay una técnica simple para eliminar pisos y techos de las recurrencias, llamada transformación de dominio.

• Primero, porque estamos derivando un límite superior, podemos sobreestimar con seguridad T(n), una vez pretendiendo que los dos tamaños de subproblema son iguales, y nuevamente para eliminar el techo⁸:

T(n) ≤ 2T(⌈n/2⌉) + n ≤ 2T(n/2 + 1) + n.

• Segundo, definimos una nueva función S(n) = T(n + α), eligiendo la constante α para que S(n) satisfaga la recurrencia más simple S(n) ≤ 2S(n/2) + O(n). Para encontrar la constante correcta α, derivamos una recurrencia para S de nuestra recurrencia dada para T:

S(n) = T(n + α) [definición de S] ≤ 2T(n/2 + α/2 + 1) + n + α [recurrencia para T]  
= 2S(n/2 − α/2 + 1) + n + α [definición de S]

Establecer α = 2 simplifica esta recurrencia a S(n) ≤ 2S(n/2) + n + 2, que es exactamente lo que queríamos.

• Finalmente, el método del árbol de recursión implica S(n) = O(n log n), y por lo tanto

T(n) = S(n − 2) = O((n − 2) log(n − 2)) = O(n log n),

exactamente como se prometió.

Se pueden usar transformaciones de dominio similares para eliminar pisos, techos e incluso términos de orden menor de cualquier recurrencia de divide y vencerás. Pero ahora que nos damos cuenta de esto, ¡no necesitamos molestarnos en analizar los detalles nunca más! De ahora en adelante, enfrentado con cualquier recurrencia de divide y vencerás, silenciosamente pasaré por alto pisos y techos y términos de orden menor, y te animo a hacer lo mismo.

**♥ 1.8 Selección en Tiempo Lineal**

Durante nuestra discusión de quicksort, afirmé de pasada que podemos encontrar la mediana de un arreglo no ordenado en tiempo lineal. El primer algoritmo de este tipo fue descubierto por Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest y Bob Tarjan a principios de los años 1970. Su algoritmo en realidad resuelve el problema más general de seleccionar el k-ésimo elemento más pequeño en un arreglo de n elementos, dados el arreglo y el entero k como entrada, usando una variante de un algoritmo llamado quickselect o quicksort de un brazo. Quickselect fue descrito por primera vez por Tony Hoare en 1961, literalmente en la misma página donde publicó por primera vez quicksort.

**Quickselect**

El algoritmo genérico quickselect elige un elemento pivote, particiona el arreglo usando la misma subrutina Partition que QuickSort, y luego busca recursivamente solo uno de los dos subarreglos, específicamente, el que contiene el k-ésimo elemento más pequeño del arreglo de entrada original. Se da pseudocódigo para quickselect en la Figura 1.12.

QuickSelect(A[1 .. n], k):

if n = 1

return A[1]

else

Choose a pivot element A[p]

r ← Partition(A[1 .. n], p)

if k < r

return QuickSelect(A[1 .. r − 1], k)

else if k > r

return QuickSelect(A[r + 1 .. n], k − r)

else

return A[r]

**Figura 1.12.** Quickselect, o quicksort de un brazo

Este algoritmo tiene dos características importantes. Primero, al igual que quicksort, la corrección de quickselect no depende de cómo se elige el pivote. Segundo, incluso si realmente solo nos importa seleccionar medianas (el caso especial k = n/2), la estrategia recursiva de Hoare requiere que consideremos el problema de selección más general; la mediana del arreglo de entrada A[1 .. n] casi nunca es la mediana de ninguno de los dos subarreglos más pequeños A[1 .. r − 1] o A[r + 1 .. n].

El tiempo de ejecución en el peor caso de QuickSelect obedece una recurrencia similar a QuickSort. No conocemos el valor de r, o cuál de los dos subarreglos buscaremos recursivamente, así que tenemos que asumir lo peor.

T(n) ≤ max₁≤r≤n max{T(r − 1), T(n − r)} + O(n)

Podemos simplificar la recurrencia ligeramente dejando que ℓ denote la longitud del subproblema recursivo:

T(n) ≤ max₀≤ℓ≤n−1 T(ℓ) + O(n)

Si el elemento pivote elegido es siempre el elemento más pequeño o más grande en el arreglo, la recurrencia se simplifica a T(n) = T(n − 1) + O(n), lo que implica T(n) = O(n²). (El árbol de recursión para esta recurrencia es solo un camino simple.)

**Buenos pivotes**

Podríamos evitar este comportamiento cuadrático en el peor caso si pudiéramos de alguna manera elegir mágicamente un buen pivote, significando ℓ ≤ αn para alguna constante α < 1. En este caso, la recurrencia se simplificaría a

T(n) ≤ T(αn) + O(n).

Esta recurrencia se expande en una serie geométrica decreciente, que está dominada por su término más grande, así que T(n) = O(n). (Nuevamente, el árbol de recursión es solo un camino simple. La constante en el tiempo de ejecución O(n) depende de la constante α.)

En otras palabras, si pudiéramos de alguna manera encontrar rápidamente un elemento que esté incluso cerca de la mediana en tiempo lineal, podríamos encontrar la mediana exacta en tiempo lineal. Así que ahora todo lo que necesitamos es un Hada de la Mediana Aproximada. El algoritmo de Blum-Floyd-Pratt-Rivest-Tarjan elige un buen pivote de quickselect calculando recursivamente la mediana de un subconjunto cuidadosamente elegido del arreglo de entrada. ¡El Hada de la Mediana Aproximada es solo el Hada de la Recursión disfrazada!

Específicamente, dividimos el arreglo de entrada en ⌈n/5⌉ bloques, cada uno conteniendo exactamente 5 elementos, excepto posiblemente el último. (Si el último bloque no está lleno, solo agrega algunos ∞s.) Calculamos la mediana de cada bloque por fuerza bruta, recolectamos esas medianas en un nuevo arreglo M[1 .. ⌈n/5⌉], y luego calculamos recursivamente la mediana de este nuevo arreglo. Finalmente, usamos la mediana de las medianas de bloques (llamada "mom" en el pseudocódigo abajo) como el pivote de quickselect.

MomSelect(A[1 .. n], k):

if n ≤ 25 《《o lo que sea》》

use brute force

else

m ← ⌈n/5⌉

for i ← 1 to m

M[i] ← MedianOfFive(A[5i − 4 .. 5i]) 《《¡Fuerza bruta!》》

mom ← MomSelect(M[1 .. m], ⌊m/2⌋) 《《¡Recursión!》》

r ← Partition(A[1 .. n], mom)

if k < r

return MomSelect(A[1 .. r − 1], k) 《《¡Recursión!》》

else if k > r

return MomSelect(A[r + 1 .. n], k − r) 《《¡Recursión!》》

else

return mom

MomSelect usa recursión para dos propósitos diferentes; la primera vez para elegir un elemento pivote (mom), y la segunda vez para buscar a través de las entradas en un lado de ese pivote.

**Análisis**

¿Pero por qué es esto rápido? La primera idea clave es que la mediana de medianas es un buen pivote. Mom es mayor que ⌊⌈n/5⌉/2⌋ − 1 ≈ n/10 medianas de bloques, y cada mediana de bloque es mayor que otros dos elementos en su bloque. Así, mom es mayor que al menos 3n/10 elementos en el arreglo de entrada; simétricamente, mom es menor que al menos 3n/10 elementos. Así, en el peor caso, la segunda llamada recursiva busca un arreglo de tamaño a lo sumo 7n/10.

Podemos visualizar el comportamiento del algoritmo dibujando el arreglo de entrada como una cuadrícula de 5 × ⌈n/5⌉, donde cada columna representa cinco elementos consecutivos. Para propósitos de ilustración, imagina que ordenamos cada columna de arriba hacia abajo, y luego ordenamos las columnas por su elemento medio. (¡Déjame enfatizar que el algoritmo no hace esto realmente!) En este arreglo, la mediana de medianas es el elemento más cercano al centro de la cuadrícula.

La mitad izquierda de las primeras tres filas de la cuadrícula contiene 3n/10 elementos, cada uno de los cuales es menor que mom. Si el elemento que estamos buscando es mayor que mom, nuestro algoritmo descartará todo lo que sea menor que mom, incluyendo esos 3n/10 elementos, antes de recurrir. Así, la entrada al subproblema recursivo contiene a lo sumo 7n/10 elementos. Un argumento simétrico implica que si nuestro elemento objetivo es menor que mom, descartamos al menos 3n/10 elementos mayores que mom, así que la entrada a nuestro subproblema recursivo tiene a lo sumo 7n/10 elementos.

Está bien, así que mom es un buen pivote, pero nuestro algoritmo aún hace dos llamadas recursivas en lugar de solo una; ¿cómo probamos tiempo lineal? La segunda idea clave es que el tamaño total de los dos subproblemas recursivos es un factor constante menor que el tamaño del arreglo de entrada original. El tiempo de ejecución en el peor caso del algoritmo obedece la recurrencia

T(n) ≤ T(n/5) + T(7n/10) + O(n).

Si dibujamos el árbol de recursión para esta recurrencia, observamos que el trabajo total en cada nivel del árbol de recursión es a lo sumo 9/10 del trabajo total en el nivel anterior. Así, las sumas de nivel decaen exponencialmente, dándonos la solución T(n) = O(n). (Nuevamente, el hecho de que el árbol de recursión esté desequilibrado es completamente inmaterial.) ¡Hurra! ¡Gracias, Mamá!

**Verificación de Cordura**

En este punto, muchos estudiantes preguntan sobre esa constante mágica 5. ¿Por qué elegimos ese tamaño de bloque particular? La respuesta es que 5 es el tamaño de bloque impar más pequeño que nos da decaimiento exponencial en el análisis del árbol de recursión! (Los tamaños de bloque pares introducen complicaciones adicionales.) Si hubiéramos usado bloques de tamaño 3 en su lugar, la recurrencia de tiempo de ejecución sería

T(n) ≤ T(n/3) + T(2n/3) + O(n).

¡Hemos visto esta recurrencia antes! Cada nivel del árbol de recursión tiene valor total a lo sumo n, y la profundidad del árbol de recursión es log₃/₂ n = O(log n), así que la solución a esta recurrencia es T(n) ≤ O(n log n). (Además, este análisis es ajustado, porque el árbol de recursión tiene log₃ n niveles completos.) La selección de mediana de medianas usando bloques de 3 elementos no es más rápida que ordenar.

Un análisis más fino revela que la constante oculta por la notación O() es bastante grande, incluso si solo contamos comparaciones. Seleccionar la mediana de 5 elementos requiere a lo sumo 6 comparaciones, así que necesitamos a lo sumo 6n/5 comparaciones para establecer el subproblema recursivo. Particionar ingenuamente el arreglo después de la llamada recursiva requeriría n − 1 comparaciones, pero ya conocemos 3n/10 elementos mayores que el pivote y 3n/10 elementos menores que el pivote, así que particionar en realidad requiere solo 2n/5 comparaciones adicionales. Así, una recurrencia más precisa para el número de comparaciones en el peor caso es

T(n) ≤ T(n/5) + T(7n/10) + 8n/5.

El método del árbol de recursión implica el límite superior

T(n) ≤ (8n/5) ∑ᵢ≥₀ (9/10)ⁱ = (8n/5) · 10 = 16n.

En la práctica, la selección de mediana de medianas no es tan lenta como este análisis de peor caso predice —obtener un pivote de peor caso en cada nivel de recursión es increíblemente improbable— pero aún es más lenta que ordenar para arreglos incluso moderadamente grandes⁹.

⁹De hecho, la forma correcta de elegir el elemento pivote en la práctica es elegirlo uniformemente al azar. Entonces el número esperado de comparaciones requeridas para encontrar la mediana es a lo sumo 4n. Ve mis notas de algoritmos aleatorios en http://algorithms.wtf para más detalles.

**1.9 Multiplicación Rápida**

En el capítulo anterior, vimos dos algoritmos antiguos para multiplicar dos números de n dígitos en tiempo O(n²): el algoritmo de lattice de escuela primaria y el algoritmo del campesino egipcio.

Tal vez podamos obtener un algoritmo más eficiente dividiendo los arreglos de dígitos por la mitad y explotando la siguiente identidad:

(10ᵐa + b)(10ᵐc + d) = 10²ᵐac + 10ᵐ(bc + ad) + bd

Esta recurrencia sugiere inmediatamente el siguiente algoritmo de divide y vencerás para multiplicar dos números de n dígitos x e y. Cada uno de los cuatro subproductos ac, bc, ad y bd se calcula recursivamente, pero las multiplicaciones en la última línea no son recursivas, porque podemos multiplicar por una potencia de diez desplazando los dígitos hacia la izquierda y llenando el número correcto de ceros, todo en tiempo O(n).

SplitMultiply(x, y, n):

if n = 1

return x · y

else

m ← ⌈n/2⌉

a ← ⌊x/10ᵐ⌋; b ← x mod 10ᵐ 《《x = 10ᵐa + b》》

c ← ⌊y/10ᵐ⌋; d ← y mod 10ᵐ 《《y = 10ᵐc + d》》

e ← SplitMultiply(a, c, m)

f ← SplitMultiply(b, d, m)

g ← SplitMultiply(b, c, m)

h ← SplitMultiply(a, d, m)

return 10²ᵐe + 10ᵐ(g + h) + f

La corrección de este algoritmo se sigue fácilmente por inducción. El tiempo de ejecución para este algoritmo sigue la recurrencia

T(n) = 4T(⌈n/2⌉) + O(n).

El método del árbol de recursión transforma esta recurrencia en una serie geométrica creciente, lo que implica T(n) = O(n^(log₂ 4)) = O(n²). De hecho, este algoritmo multiplica cada dígito de x con cada dígito de y, igual que el algoritmo de lattice.

Así que supongo que eso no funcionó. Muy mal. Era una buena idea.

A mediados de los años 1950, Andrei Kolmogorov, uno de los gigantes de las matemáticas del siglo XX, conjeturó públicamente que no hay algoritmo para multiplicar dos números de n dígitos en tiempo subcuadrático. Kolmogorov organizó un seminario en la Universidad de Moscú en 1960, donde reafirmó su "conjetura n²" y planteó varios problemas relacionados que planeaba discutir en futuras reuniones. Casi exactamente una semana después, un estudiante de 23 años llamado Anatolii Karatsuba presentó a Kolmogorov un contraejemplo notable. Según el propio Karatsuba,

Después del seminario le dije a Kolmogorov sobre el nuevo algoritmo y sobre la refutación de la conjetura n². Kolmogorov estaba muy agitado porque esto contradecía su conjetura muy plausible. En la siguiente reunión del seminario, Kolmogorov mismo les dijo a los participantes sobre mi método, y en ese punto el seminario fue terminado.

Karatsuba observó que el coeficiente medio bc + ad puede calcularse de los otros dos coeficientes ac y bd usando solo una multiplicación recursiva más, a través de la siguiente identidad algebraica:

ac + bd − (a − b)(c − d) = bc + ad

Este truco nos permite reemplazar las cuatro llamadas recursivas en el algoritmo anterior con solo tres llamadas recursivas, como se muestra abajo:

FastMultiply(x, y, n):

if n = 1

return x · y

else

m ← ⌈n/2⌉

a ← ⌊x/10ᵐ⌋; b ← x mod 10ᵐ 《《x = 10ᵐa + b》》

c ← ⌊y/10ᵐ⌋; d ← y mod 10ᵐ 《《y = 10ᵐc + d》》

e ← FastMultiply(a, c, m)

f ← FastMultiply(b, d, m)

g ← FastMultiply(a − b, c − d, m)

return 10²ᵐe + 10ᵐ(e + f − g) + f

El tiempo de ejecución del algoritmo FastMultiply de Karatsuba sigue la recurrencia

T(n) ≤ 3T(⌈n/2⌉) + O(n)

Una vez más, el método del árbol de recursión transforma esta recurrencia en una serie geométrica creciente, pero la nueva solución es solo T(n) = O(n^(log₂ 3)) = O(n^1.58496), una mejora significativa sobre nuestro límite de tiempo cuadrático anterior¹⁰.

El algoritmo de Karatsuba posiblemente lanzó el diseño y análisis de algoritmos como un campo formal de estudio.

Podemos llevar la idea de Karatsuba aún más lejos, dividiendo los números en más piezas y combinándolos de maneras más complicadas, para obtener algoritmos de multiplicación aún más rápidos. Andrei Toom descubrió una familia infinita de algoritmos que dividen cualquier entero en k partes, cada una con n/k dígitos, y luego calculan el producto usando solo 2k − 1 multiplicaciones recursivas; los algoritmos de Toom fueron simplificados más por Stephen Cook en su tesis doctoral. Para cualquier k fijo, el algoritmo Toom-Cook se ejecuta en tiempo O(n^(1+1/(lg k))), donde la constante oculta en la notación O(·) depende de k.

Finalmente, esta estrategia de divide y vencerás llevó a Gauss (sí, realmente) al descubrimiento de la transformada rápida de Fourier¹¹. El algoritmo FFT básico mismo se ejecuta en tiempo O(n log n); sin embargo, usar FFTs para multiplicación de enteros incurre en algún overhead adicional pequeño. El primer algoritmo de multiplicación de enteros basado en FFT, publicado por Arnold Schönhage y Volker Strassen en 1971, se ejecuta en tiempo O(n log n log log n). Schönhage-Strassen permaneció como el algoritmo de multiplicación de enteros teóricamente más rápido durante varias décadas, antes de que Martin Fürer descubriera la primera de una larga serie de mejoras técnicas. Finalmente, en 2019, David Harvey y Joris van der Hoeven publicaron un algoritmo que se ejecuta en tiempo O(n log n)¹².

**1.10 Exponenciación**

Dados un número a y un entero positivo n, supongamos que queremos calcular aⁿ. El método ingenuo estándar es un bucle for simple que realiza n − 1 multiplicaciones por a:

SlowPower(a, n):

x ← a

for i ← 2 to n

x ← x · a

return x

Este algoritmo iterativo requiere n multiplicaciones.

El parámetro de entrada a podría ser un entero, o un racional, o un número de punto flotante. De hecho, no necesita ser un número en absoluto, siempre que sea algo que sepamos cómo multiplicar. Por ejemplo, el mismo algoritmo puede usarse para calcular potencias módulo algún número finito (una operación comúnmente usada en algoritmos de criptografía) o para calcular potencias de matrices (una operación usada para evaluar recurrencias y calcular caminos más cortos en grafos). Porque no sabemos qué tipo de objeto estamos multiplicando, no podemos saber cuánto tiempo requiere una sola multiplicación, así que nos vemos forzados a analizar el tiempo de ejecución en términos del número de multiplicaciones.

Hay un método de divide y vencerás mucho más rápido, originalmente propuesto por el prosodista indio Piṅgala en el siglo II aec, que usa la siguiente fórmula recursiva simple:

aⁿ = { 1 si n = 0 (aⁿ/²)² si n > 0 y n es par (a⌊ⁿ/²⌋)² · a de lo contrario }

PiṅgalaPower(a, n):

if n = 1

return a

else

x ← PiṅgalaPower(a, ⌊n/2⌋)

if n is even

return x · x

else

return x · x · a

El número total de multiplicaciones realizadas por este algoritmo satisface la recurrencia T(n) ≤ T(n/2) + 2. El método del árbol de recursión inmediatamente nos da la solución T(n) = O(log n).

Un algoritmo de exponenciación casi idéntico también puede derivarse directamente del algoritmo de multiplicación del campesino egipcio del capítulo anterior, reemplazando suma con multiplicación (y en particular, reemplazando duplación con cuadrado).

aⁿ = { 1 si n = 0 (a²)ⁿ/² si n > 0 y n es par (a²)⌊ⁿ/²⌋ · a de lo contrario }

PeasantPower(a, n):

if n = 1

return a

else if n is even

return PeasantPower(a², n/2)

else

return PeasantPower(a², ⌊n/2⌋) · a

Este algoritmo —que podría llamarse razonablemente "cuadrado y mediación"— también realiza solo O(log n) multiplicaciones.

Ambos algoritmos son asintóticamente óptimos; cualquier algoritmo que calcule aⁿ debe realizar al menos Ω(log n) multiplicaciones, porque cada multiplicación a lo sumo duplica la potencia más grande calculada hasta ahora. De hecho, cuando n es una potencia de dos, ambos algoritmos requieren exactamente log₂ n multiplicaciones, lo que es exactamente óptimo. Sin embargo, hay métodos ligeramente más rápidos para otros valores de n. Por ejemplo, PiṅgalaPower y PeasantPower cada uno calcula a¹⁵ usando seis multiplicaciones, pero de hecho solo cinco multiplicaciones son necesarias:

• Piṅgala: a → a² → a³ → a⁶ → a⁷ → a¹⁴ → a¹⁵ • Campesino: a → a² → a⁴ → a⁸ → a¹² → a¹⁴ → a¹⁵  
• Óptimo: a → a² → a³ → a⁵ → a¹⁰ → a¹⁵

Es una pregunta abierta de larga data si el número absoluto mínimo de multiplicaciones para un exponente dado n puede calcularse eficientemente.

**EJERCICIOS**

**Torre de Hanoi**

1. Demuestra que el algoritmo recursivo original de la Torre de Hanoi realiza exactamente la misma secuencia de movimientos —los mismos discos, hacia y desde las mismas clavijas, en el mismo orden— que cada uno de los siguientes algoritmos no recursivos. Las clavijas están etiquetadas 0, 1 y 2, y nuestro problema es mover una pila de n discos desde la clavija 0 a la clavija 2 (como se muestra en la página 24).

(a) Si n es par, intercambia las clavijas 1 y 2. En el i-ésimo paso, haz el único movimiento legal que evite la clavija i mod 3. Si no hay movimiento legal, entonces todos los discos están en la clavija i mod 3, y el rompecabezas está resuelto.

(b) Para el primer movimiento, mueve el disco 1 a la clavija 1 si n es par y a la clavija 2 si n es impar. Luego haz repetidamente el único movimiento legal que involucre un disco diferente del movimiento anterior. Si no existe tal movimiento, el rompecabezas está resuelto.

(c) Pretende que los discos n + 1, n + 2 y n + 3 están en el fondo de las clavijas 0, 1 y 2, respectivamente. Haz repetidamente el único movimiento legal que satisfaga las siguientes restricciones, hasta que no sea posible tal movimiento.

• No coloques un disco impar directamente encima de otro disco impar. • No coloques un disco par directamente encima de otro disco par. • No deshagas el movimiento anterior.

(d) Sea ρ(n) el entero más pequeño k tal que n/2ᵏ no es un entero. Por ejemplo, ρ(42) = 2, porque 42/2¹ es un entero pero 42/2² no lo es. (Equivalentemente, ρ(n) es uno más que la posición del 1 menos significativo en la representación binaria de n.) Debido a que su comportamiento se asemeja a las marcas en una regla, ρ(n) a veces se llama la función regla.

RulerHanoi(n):

i ← 1

while ρ(i) ≤ n

if n − i is even

move disk ρ(i) forward 《《0 → 1 → 2 → 0》》

else

move disk ρ(i) backward 《《0 → 2 → 1 → 0》》

i ← i + 1

1. La Torre de Hanoi es un descendiente relativamente reciente de un rompecabezas mecánico mucho más antiguo conocido como los anillos chinos enlazados, Baguenaudier, Anillos de Cardan, Meleda, Paciencia, Hierros Fatigantes, Cerradura del Prisionero, Spin-Out, y muchos otros nombres. Este rompecabezas ya era bien conocido tanto en China como en Europa en el siglo XVI. El matemático italiano Luca Pacioli describió el rompecabezas de 7 anillos y su solución en su tratado no publicado De Viribus Quantitatis, escrito entre 1498 y 1506¹³; solo unos pocos años después, el poeta de la dinastía Ming Yang Shen describió el rompecabezas de 9 anillos como "un juguete para mujeres y niños". El rompecabezas se atribuye apocríficamente a un general chino del siglo II, quien le dio el rompecabezas a su esposa para ocupar su tiempo mientras él estaba en la guerra.

**Figura 1.16.** El Baguenaudier de 7 anillos, de Récréations Mathématiques por Édouard Lucas (1891) (Ver Créditos de Imagen al final del libro.)

El rompecabezas Baguenaudier tiene muchas formas físicas, pero una de las más comunes consiste en un bucle de metal largo y varios anillos, que están conectados a una base sólida por varillas móviles. El bucle inicialmente se enhebra a través de los anillos como se muestra en la Figura 1.16; el objetivo del rompecabezas es quitar el bucle.

Más abstractamente, podemos modelar el rompecabezas como una secuencia de bits, uno para cada anillo, donde el i-ésimo bit es 1 si el bucle pasa a través del i-ésimo anillo y 0 de lo contrario. (Aquí indexamos los anillos de derecha a izquierda, como se muestra en la Figura 1.16.) El rompecabezas permite dos movimientos legales:

• Siempre puedes voltear el 1er (= más a la derecha) bit. • Si la cadena de bits termina con exactamente z 0s, puedes voltear el (z + 2)-ésimo bit.

El objetivo del rompecabezas es transformar una cadena de n 1s en una cadena de n 0s.

Por ejemplo, la siguiente secuencia de 21 movimientos resuelve el rompecabezas de 5 anillos:

11111 →¹ 11110 →³ 11010 →¹ 11011 →² 11001 →¹ 11000  
→⁵ 01000 →¹ 01001 →² 01011 →³ 01111 →¹ 01110  
→² 01010 →¹ 01011 →⁴ 00011 →¹ 00010 →² 00000  
→³ 00100 →¹ 00101 →² 00111 →¹ 00110 →¹ 00100 →¹ 00000 ♦

(a) Llama a una secuencia de movimientos reducida si ningún movimiento es el inverso del movimiento anterior. Demuestra que para cualquier entero no negativo n, hay exactamente una secuencia reducida de movimientos que resuelve el rompecabezas Baguenaudier de n anillos. [Pista: Este problema es mucho más fácil si ya estás familiarizado con grafos.]

(b) Describe un algoritmo para resolver el rompecabezas Baguenaudier. Tu entrada es el número de anillos n; tu algoritmo debería imprimir una secuencia reducida de movimientos que resuelva el rompecabezas. Por ejemplo, dado el entero 5 como entrada, tu algoritmo debería imprimir la secuencia 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1.

(c) ¿Exactamente cuántos movimientos realiza tu algoritmo, como función de n? Demuestra que tu respuesta es correcta.

1. Un capítulo menos familiar en la historia de la Torre de Hanoi es su breve reubicación del templo de Benares a Pisa a principios del siglo XIII¹⁴. La reubicación fue organizada por el rico comerciante-matemático Leonardo Fibonacci, a pedido del Emperador del Sacro Imperio Romano Federico II, quien había escuchado informes del templo de soldados que regresaban de las Cruzadas. Las Torres de Pisa y sus monjes asistentes se hicieron famosos, ayudando a establecer a Pisa como un centro comercial dominante en la península italiana.

Desafortunadamente, casi tan pronto como el templo fue movido, una de las agujas de diamante comenzó a inclinarse hacia un lado. Para evitar la posibilidad de que la torre inclinada se cayera por demasiado uso, Fibonacci convenció a los sacerdotes de adoptar una regla más relajada: Cualquier número de discos en la aguja inclinada pueden moverse juntos a otra aguja en un solo movimiento. Seguía estando prohibido colocar un disco más grande encima de un disco más pequeño, y los discos tenían que moverse uno a la vez hacia la aguja inclinada o entre las dos agujas verticales.

**Figura 1.17.** Las Torres de Pisa. En el quinto movimiento, dos discos se quitan de la aguja inclinada.

Gracias a la nueva regla de Fibonacci, los sacerdotes podían provocar el fin del universo algo más rápido desde Pisa de lo que podían desde Benares. Afortunadamente, el templo fue movido de Pisa de vuelta a Benares después de que el recién coronado Papa Gregorio IX excomulgara a Federico II, haciendo que los sacerdotes locales fueran menos comprensivos a hospedar herejes extranjeros con hábitos matemáticos extraños. Poco después, se erigió una torre campanario en el lugar donde una vez estuvo el templo; también comenzó a inclinarse casi inmediatamente.

Describe un algoritmo para transferir una pila de n discos de una aguja vertical a la otra aguja vertical, usando el menor número posible de movimientos. ¿Exactamente cuántos movimientos realiza tu algoritmo?

1. Considera las siguientes variantes restringidas del rompecabezas de la Torre de Hanoi. En cada problema, las clavijas están numeradas 0, 1 y 2, y tu tarea es mover una pila de n discos desde la clavija 0 a la clavija 2, exactamente como en el problema 1.

(a) Supón que tienes prohibido mover cualquier disco directamente entre la clavija 1 y la clavija 2; cada movimiento debe involucrar la clavija 0. Describe un algoritmo para resolver esta versión del rompecabezas en tan pocos movimientos como sea posible. ¿Exactamente cuántos movimientos hace tu algoritmo?

♣♥(b) Supón que solo puedes mover discos de la clavija 0 a la clavija 2, de la clavija 2 a la clavija 1, o de la clavija 1 a la clavija 0. Equivalentemente, supón que las clavijas están dispuestas en círculo y numeradas en orden horario, y solo puedes mover discos en sentido antihorario. Describe un algoritmo para resolver esta versión del rompecabezas en tan pocos movimientos como sea posible. ¿Cuántos movimientos hace tu algoritmo?

♣♥(c) Finalmente, supón que tu única restricción es que nunca puedes mover un disco directamente de la clavija 0 a la clavija 2. Describe un algoritmo para resolver esta versión del rompecabezas en tan pocos movimientos como sea posible. ¿Cuántos movimientos hace tu algoritmo? [Pista: ¡Matrices! Esta variante es considerablemente más difícil de analizar que las otras dos.]

1. Considera la siguiente variante más compleja del rompecabezas de la Torre de Hanoi. El rompecabezas tiene una fila de k clavijas, numeradas del 1 al k. En un solo turno, puedes mover el disco más pequeño en la clavija i a la clavija i − 1 o a la clavija i + 1, para cualquier índice i; como es usual, no puedes colocar un disco más grande en un disco más pequeño. Tu misión es mover una pila de n discos desde la clavija 1 a la clavija k.

(a) Describe un algoritmo recursivo para el caso k = 3. ¿Exactamente cuántos movimientos hace tu algoritmo? (Esto es exactamente lo mismo que el problema 4(a).)

(b) Describe un algoritmo recursivo para el caso k = n + 1 que requiera a lo sumo O(n³) movimientos. [Pista: Usa la parte (a).]

♥(c) Describe un algoritmo recursivo para el caso k = n + 1 que requiera a lo sumo O(n²) movimientos. [Pista: No uses la parte (a).]

♥(d) Describe un algoritmo recursivo para el caso k = √n que requiera a lo sumo un número polinomial de movimientos. (¿Cuál polinomio??)

♥(e) Describe y analiza un algoritmo recursivo para n y k arbitrarios. ¿Qué tan pequeño debe ser k (como función de n) para que el número de movimientos esté acotado por un polinomio en n?

**Árboles de Recursión**

1. Usa árboles de recursión para resolver cada una de las siguientes recurrencias.

A(n) = 2A(n/4) + √n B(n) = 2B(n/4) + n C(n) = 2C(n/4) + n²  
D(n) = 3D(n/3) + √n E(n) = 3E(n/3) + n F(n) = 3F(n/3) + n²  
G(n) = 4G(n/2) + √n H(n) = 4H(n/2) + n I(n) = 4I(n/2) + n²

1. Usa árboles de recursión para resolver cada una de las siguientes recurrencias.

(j) J(n) = J(n/2) + J(n/3) + J(n/6) + n  
(k) K(n) = K(n/2) + 2K(n/3) + 3K(n/4) + n²  
(l) L(n) = L(n/15) + L(n/10) + 2L(n/6) + √n

♥8. Usa árboles de recursión para resolver cada una de las siguientes recurrencias.

(m) M(n) = 2M(n/2) + O(n log n)  
(n) N(n) = 2N(n/2) + O(n/ log n)  
(p) P(n) = √n P(√n) + n  
(q) Q(n) = √2n Q(√2n) + √n

**Ordenamiento**

1. Supón que te dan una pila de n panqueques de diferentes tamaños. Quieres ordenar los panqueques para que los panqueques más pequeños estén encima de los panqueques más grandes. La única operación que puedes realizar es un volteo —insertar una espátula bajo los k panqueques superiores, para algún entero k entre 1 y n, y voltearlos todos.

**Figura 1.19.** Volteando los cuatro panqueques superiores.

(a) Describe un algoritmo para ordenar una pila arbitraria de n panqueques usando O(n) volteos. ¿Exactamente cuántos volteos realiza tu algoritmo en el peor caso?¹⁵ [Pista: Este problema no tiene nada que ver con la Torre de Hanoi.]

(b) Para cada entero positivo n, describe una pila de n panqueques que requiera Ω(n) volteos para ordenar.

(c) Ahora supón que un lado de cada panqueque está quemado. Describe un algoritmo para ordenar una pila arbitraria de n panqueques, para que el lado quemado de cada panqueque mire hacia abajo, usando O(n) volteos. ¿Exactamente cuántos volteos realiza tu algoritmo en el peor caso?

1. Recuerda que la heurística mediana de tres examina el primer, último y elemento medio del arreglo, y usa la mediana de esos tres elementos como pivote de quicksort. Demuestra que quicksort con la heurística mediana de tres requiere tiempo Ω(n²) para ordenar un arreglo de tamaño n en el peor caso. Específicamente, para cualquier entero n, describe una permutación de los enteros 1 hasta n, tal que en cada llamada recursiva a mediana-de-tres-quicksort, el pivote sea siempre el segundo elemento más pequeño del arreglo. Diseñar esta permutación requiere conocimiento íntimo de la subrutina Partition.

(a) Como ejercicio de calentamiento, asume que la subrutina Partition es estable, significando que preserva el orden existente de todos los elementos menores que el pivote, y preserva el orden existente de todos los elementos menores que el pivote.

♥(b) Asume que la subrutina Partition usa el algoritmo específico listado en la página 29, que no es estable.

1. (a) ¡Oye, Moe! ¡Oye, Larry! ¡Demuestra que el siguiente algoritmo realmente ordena su entrada!

StoogeSort(A[0 .. n − 1]):

if n = 2 and A[0] > A[1]

swap A[0] ↔ A[1]

else if n > 2

m = ⌈2n/3⌉

StoogeSort(A[0 .. m − 1])

StoogeSort(A[n − m .. n − 1])

StoogeSort(A[0 .. m − 1])

(b) ¿StoogeSort seguiría ordenando correctamente si reemplazáramos m = ⌈2n/3⌉ con m = ⌊2n/3⌋? Justifica tu respuesta.

(c) Establece una recurrencia (incluyendo el(los) caso(s) base) para el número de comparaciones ejecutadas por StoogeSort.

(d) Resuelve la recurrencia, y demuestra que tu solución es correcta. [Pista: Ignora el techo.]

(e) Demuestra que el número de intercambios ejecutados por StoogeSort es a lo sumo (n₂).

1. El siguiente algoritmo de ordenamiento cruel e inusual fue propuesto por Gary Miller:

Cruel(A[1 .. n]):

if n > 1

Cruel(A[1 .. n/2])

Cruel(A[n/2 + 1 .. n])

Unusual(A[1 .. n])

Unusual(A[1 .. n]):

if n = 2

if A[1] > A[2] 《《¡la única comparación!》》

swap A[1] ↔ A[2]

else

for i ← 1 to n/4 《《intercambiar 2do y 3er cuartos》》

swap A[i + n/4] ↔ A[i + n/2]

Unusual(A[1 .. n/2]) 《《recurrir en la mitad izquierda》》

Unusual(A[n/2 + 1 .. n]) 《《recurrir en la mitad derecha》》

Unusual(A[n/4 + 1 .. 3n/4]) 《《recurrir en la mitad media》》

Las comparaciones realizadas por este algoritmo no dependen en absoluto de los valores en el arreglo de entrada; tal algoritmo de ordenamiento se llama inconsciente. Asume para este problema que el tamaño de entrada n es siempre una potencia de 2.

(a) Demuestra por inducción que Cruel ordena correctamente cualquier arreglo de entrada. [Pista: Considera un arreglo que contenga n/4 1s, n/4 2s, n/4 3s y n/4 4s. ¿Por qué es suficiente este caso especial?]

(b) Demuestra que Cruel no ordenaría correctamente si removiéramos el bucle for de Unusual.

(c) Demuestra que Cruel no ordenaría correctamente si intercambiáramos las últimas dos líneas de Unusual.

(d) ¿Cuál es el tiempo de ejecución de Unusual? Justifica tu respuesta.

(e) ¿Cuál es el tiempo de ejecución de Cruel? Justifica tu respuesta.

1. Una inversión en un arreglo A[1 .. n] es un par de índices (i, j) tal que i < j y A[i] > A[j]. El número de inversiones en un arreglo de n elementos está entre 0 (si el arreglo está ordenado) y (n₂) (si el arreglo está ordenado al revés). Describe y analiza un algoritmo para contar el número de inversiones en un arreglo de n elementos en tiempo O(n log n). [Pista: Modifica mergesort.]
2. (a) Supón que te dan dos conjuntos de n puntos, un conjunto {p₁, p₂, ..., pₙ} en la línea y = 0 y el otro conjunto {q₁, q₂, ..., qₙ} en la línea y = 1. Crea un conjunto de n segmentos de línea conectando cada punto pᵢ al punto correspondiente qᵢ. Describe y analiza un algoritmo de divide y vencerás para determinar cuántos pares de estos segmentos de línea se intersectan, en tiempo O(n log n). [Pista: Ve el problema anterior.]

(b) Ahora supón que te dan dos conjuntos {p₁, p₂, ..., pₙ} y {q₁, q₂, ..., qₙ} de n puntos en el círculo unitario. Conecta cada punto pᵢ al punto correspondiente qᵢ. Describe y analiza un algoritmo de divide y vencerás para determinar cuántos pares de estos segmentos de línea se intersectan en tiempo O(n log² n). [Pista: Usa tu solución a la parte (a).]

♥(c) Describe un algoritmo para la parte (b) que se ejecute en tiempo O(n log n). [Pista: ¡Usa tu solución de la parte (b)!]

**Figura 1.20.** Once pares de segmentos que se intersectan con extremos en líneas paralelas, y diez pares de segmentos que se intersectan con extremos en un círculo.

1. (a) Describe un algoritmo que ordene un arreglo de entrada A[1 .. n] llamando a una subrutina SqrtSort(k), que ordena el subarreglo A[k + 1 .. k + √n] in situ, dado un entero arbitrario k entre 0 y n − √n como entrada. (Para simplificar el problema, asume que √n es un entero.) Tu algoritmo solo puede inspeccionar o modificar el arreglo de entrada llamando a SqrtSort; en particular, tu algoritmo no debe comparar, mover o copiar elementos del arreglo directamente. ¿Cuántas veces llama tu algoritmo a SqrtSort en el peor caso?

♣(b) Demuestra que tu algoritmo de la parte (a) es óptimo hasta factores constantes. En otras palabras, si f(n) es el número de veces que tu algoritmo llama a SqrtSort, demuestra que ningún algoritmo puede ordenar usando o(f(n)) llamadas a SqrtSort.

(c) Ahora supón que SqrtSort está implementado recursivamente, llamando a tu algoritmo de ordenamiento de la parte (a). Por ejemplo, en el segundo nivel de recursión, el algoritmo está ordenando arreglos de tamaño aproximadamente n^(1/4). ¿Cuál es el tiempo de ejecución en el peor caso del algoritmo de ordenamiento resultante? (Para simplificar el análisis, asume que el tamaño del arreglo n tiene la forma 2^(2^k), para que las raíces cuadradas repetidas sean siempre enteros.)

**Selección**

1. Supón que nos dan un conjunto S de n elementos, cada uno con un valor y un peso. Para cualquier elemento x ∈ S, definimos dos subconjuntos

• S<x es el conjunto de elementos de S cuyo valor es menor que el valor de x. • S>x es el conjunto de elementos de S cuyo valor es mayor que el valor de x.

Para cualquier subconjunto R ⊆ S, sea w(R) la suma de los pesos de los elementos en R. La mediana ponderada de R es cualquier elemento x tal que w(S<x) ≤ w(S)/2 y w(S>x) ≤ w(S)/2.

Describe y analiza un algoritmo para calcular la mediana ponderada de un conjunto ponderado dado en tiempo O(n). Tu entrada consiste en dos arreglos no ordenados S[1 .. n] y W[1 .. n], donde para cada índice i, el i-ésimo elemento tiene valor S[i] y peso W[i]. Puedes asumir que todos los valores son distintos y todos los pesos son positivos.

1. (a) Describe un algoritmo para determinar en tiempo O(n) si un arreglo arbitrario A[1 .. n] contiene más de n/4 copias de cualquier valor.

(b) Describe y analiza un algoritmo para determinar, dado un arreglo arbitrario A[1 .. n] y un entero k, si A contiene más de k copias de cualquier valor. Expresa el tiempo de ejecución de tu algoritmo como función de n y k.

No uses hashing, o radix sort, o cualquier otro método que dependa de los valores precisos de entrada, en oposición a su orden.

1. Describe un algoritmo para calcular la mediana de un arreglo A[1 .. 5] de números distintos usando a lo sumo 6 comparaciones. En lugar de escribir pseudocódigo, describe tu algoritmo usando un árbol de decisión: Un árbol binario donde cada nodo interno contiene una comparación de la forma "A[i] ≷ A[j]?" y cada hoja contiene un índice en el arreglo.

**Figura 1.21.** Encontrando la mediana de un arreglo de 3 elementos usando a lo sumo 3 comparaciones

1. Considera la generalización del algoritmo MomSelect de Blum-Floyd-Pratt-Rivest-Tarjan mostrado en la Figura 1.22, que particiona el arreglo de entrada en ⌈n/b⌉ bloques de tamaño b, en lugar de ⌈n/5⌉ bloques de tamaño 5, pero es idéntico en todo lo demás.

MombSelect(A[1 .. n], k):

if n ≤ b²

use brute force

else

m ← ⌈n/b⌉

for i ← 1 to m

M[i] ← MedianOfB(A[b(i − 1) + 1 .. bi])

momb ← MombSelect(M[1 .. m], ⌊m/2⌋)

r ← Partition(A[1 .. n], momb)

if k < r

return MombSelect(A[1 .. r − 1], k)

else if k > r

return MombSelect(A[r + 1 .. n], k − r)

else

return momb

**Figura 1.22.** Una familia parametrizada de algoritmos de selección; ver problema 19.

(a) Establece una recurrencia para el tiempo de ejecución de MombSelect, asumiendo que b es una constante (para que la subrutina MedianOfB se ejecute en tiempo O(1)). En particular, ¿cómo dependen los tamaños de los subproblemas recursivos de la constante b? Considera b par y b impar por separado.

(b) ¿Cuál es el tiempo de ejecución en el peor caso de Mom1Select? [Pista: Esta es una pregunta trampa.]

♣♥(c) ¿Cuál es el tiempo de ejecución en el peor caso de Mom2Select? [Pista: ¡Esta es una pregunta injusta!]

♥(d) ¿Cuál es el tiempo de ejecución en el peor caso de Mom3Select? Encontrar un límite superior en el tiempo de ejecución es directo; la parte difícil es mostrar que este análisis es realmente ajustado. [Pista: Ver problema 10.]

♥(e) ¿Cuál es el tiempo de ejecución en el peor caso de Mom4Select? Nuevamente, la parte difícil es mostrar que el análisis no puede mejorarse¹⁶.

(f) Para cualquier constante b ≥ 5, el algoritmo MombSelect se ejecuta en tiempo O(n), pero diferentes valores de b llevan a diferentes factores constantes. Sea M(b) el número mínimo de comparaciones requeridas para encontrar la mediana de b números. El valor exacto de M(b) se conoce solo para b ≤ 13:

| **b** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M(b) | 0 | 1 | 3 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 23 |

Para cada b entre 5 y 13, encuentra un límite superior en el tiempo de ejecución de MombSelect de la forma T(n) ≤ αbn para alguna constante explícita αb. (Por ejemplo, en la página 39 mostramos que α₅ ≤ 16.)

(g) ¿Qué valor de b produce la constante más pequeña αb? [Pista: Esta es una pregunta trampa!]

1. Demuestra que la variante del algoritmo Select de Blum-Floyd-Pratt-Rivest-Tarjan mostrada en la Figura 1.23, que usa una capa extra de medianas pequeñas para elegir el pivote principal, se ejecuta en tiempo O(n).

MomomSelect(A[1 .. n], k):

if n ≤ 81

use brute force

else

m ← ⌈n/3⌉

for i ← 1 to m

M[i] ← MedianOf3(A[3i − 2 .. 3i])

mm ← ⌈m/3⌉

for j ← 1 to mm

Mom[j] ← MedianOf3(M[3j − 2 .. 3j])

momom ← MomomSelect(Mom[1 .. mm], ⌊mm/2⌋)

r ← Partition(A[1 .. n], momom)

if k < r

return MomomSelect(A[1 .. r − 1], k)

else if k > r

return MomomSelect(A[r + 1 .. n], k − r)

else

return momom

**Figura 1.23.** Selección por mediana de mamás; ver problema 20).

1. (a) Supón que nos dan dos arreglos ordenados A[1 .. n] y B[1 .. n]. Describe un algoritmo para encontrar el elemento mediano en la unión de A y B en tiempo Θ(log n). Puedes asumir que los arreglos no contienen elementos duplicados.

(b) Supón que nos dan dos arreglos ordenados A[1 .. m] y B[1 .. n] y un entero k. Describe un algoritmo para encontrar el k-ésimo elemento más pequeño en A ∪ B en tiempo Θ(log(m + n)). Por ejemplo, si k = 1, tu algoritmo debería devolver el elemento más pequeño de A ∪ B.) [Pista: Usa tu solución a la parte (a).]

♥(c) Ahora supón que nos dan tres arreglos ordenados A[1 .. n], B[1 .. n] y C[1 .. n], y un entero k. Describe un algoritmo para encontrar el k-ésimo elemento más pequeño en A ∪ B ∪ C en tiempo O(log n).

(d) Finalmente, supón que nos dan un arreglo bidimensional A[1 .. m, 1 .. n] en el cual cada fila A[i,·] está ordenada, y un entero k. Describe un algoritmo para encontrar el k-ésimo elemento más pequeño en A tan rápido como sea posible. ¿Cómo depende el tiempo de ejecución de tu algoritmo de m? [Pista: Resuelve el problema 16 primero.]

**Aritmética**

1. En 1854, arqueólogos descubrieron tablillas de arcilla sumerias, talladas alrededor del 2000 aec, que listan los cuadrados de enteros hasta 59. Este descubrimiento llevó a algunos académicos a conjeturar que los antiguos sumerios realizaban multiplicación por reducción a cuadrado, usando una identidad como x · y = (x² + y² − (x − y)²)/2. Desafortunadamente, esos mismos académicos guardan silencio sobre cómo los sumerios supuestamente elevaban al cuadrado números más grandes. Cuatro mil años después, ¡finalmente podemos rescatar a estos matemáticos sumerios de sus vidas de trabajo pesado a través del poder de la recursión!

(a) Describe una variante del algoritmo de Karatsuba que eleva al cuadrado cualquier número de n dígitos en tiempo O(n^(lg 3)), reduciendo a elevar al cuadrado tres números de ⌈n/2⌉ dígitos. (Karatsuba realmente hizo esto en 1960.)

(b) Describe un algoritmo recursivo que eleva al cuadrado cualquier número de n dígitos en tiempo O(n^(log₃ 6)), reduciendo a elevar al cuadrado seis números de ⌈n/3⌉ dígitos.

♥(c) Describe un algoritmo recursivo que eleva al cuadrado cualquier número de n dígitos en tiempo O(n^(log₃ 5)), reduciendo a elevar al cuadrado solo cinco números de (n/3 + O(1)) dígitos. [Pista: ¿Qué es (a + b + c)² + (a − b + c)²?]

1. (a) Describe y analiza una variante del algoritmo de Karatsuba que multiplica cualquier número de m dígitos y cualquier número de n dígitos, para cualquier n ≥ m, en tiempo O(n m^(lg 3−1)).

(b) Describe un algoritmo para calcular la representación decimal de 2ⁿ en tiempo O(n^(lg 3)), usando el algoritmo de la parte (a) como subrutina. (El algoritmo estándar que calcula un dígito a la vez requiere tiempo Θ(n²).)

(c) Describe un algoritmo de divide y vencerás para calcular la representación decimal de un número binario arbitrario de n bits en tiempo O(n^(lg 3)). [Pista: Cuidado con un factor log extra en el tiempo de ejecución.]

♥(d) Supón que podemos multiplicar dos números de n dígitos en tiempo O(M(n)). Describe un algoritmo para calcular la representación decimal de un número binario arbitrario de n bits en tiempo O(M(n) log n). [Pista: El análisis es la parte difícil; usa una transformación de dominio.]

1. Considera el siguiente algoritmo recursivo clásico para calcular el factorial n! de un entero no negativo n:

Factorial(n):

if n = 0

return 1

else

return n · Factorial(n − 1)

(a) ¿Cuántas multiplicaciones realiza este algoritmo?

(b) ¿Cuántos bits se requieren para escribir n! en binario? Expresa tu respuesta en la forma Θ(f(n)), para alguna función familiar f(n). [Pista: (n/2)^(n/2) < n! < n^n.]

(c) Tu respuesta a (b) debería convencerte de que el número de multiplicaciones no es una buena estimación del tiempo real de ejecución de Factorial. Podemos multiplicar cualquier número de k dígitos y cualquier número de l dígitos en tiempo O(k · l) usando el algoritmo de lattice o duplación y mediación. ¿Cuál es el tiempo de ejecución de Factorial si usamos este algoritmo de multiplicación como subrutina?

(d) El siguiente algoritmo recursivo también calcula la función factorial, pero usando una agrupación diferente de las multiplicaciones:

Falling(n, m): 《《Calcular n!/(n − m)!》》

if m = 0

return 1

else if m = 1

return n

else

return Falling(n, ⌊m/2⌋) · Falling(n − ⌊m/2⌋, ⌈m/2⌉)

¿Cuál es el tiempo de ejecución de Falling(n, n) si usamos multiplicación de escuela primaria? [Pista: Como es usual, ignora los pisos y techos.]

(e) Describe y analiza una variante del algoritmo de Karatsuba que multiplica cualquier número de k dígitos y cualquier número de l dígitos, para cualquier k ≥ l, en tiempo O(k · l^(lg 3−1)) = O(k · l^0.585).

♥(f) ¿Cuáles son los tiempos de ejecución de Factorial(n) y Falling(n, n) si usamos la multiplicación Karatsuba modificada de la parte (e)?

1. El máximo común divisor de dos enteros positivos x e y, denotado gcd(x, y), es el entero más grande d tal que tanto x/d como y/d son enteros. Los Elementos de Euclides, escritos alrededor del 300 aec, describe el siguiente algoritmo recursivo para calcular gcd(x, y)¹⁷:

EuclidGCD(x, y):

if x = y

return x

else if x > y

return EuclidGCD(x − y, y)

else

return EuclidGCD(x, y − x)

(a) Demuestra que EuclidGCD calcula correctamente gcd(x, y)¹⁸. Específicamente: i. Demuestra que EuclidGCD(x, y) divide tanto x como y. ii. Demuestra que cada divisor de x e y es un divisor de EuclidGCD(x, y).

(b) ¿Cuál es el tiempo de ejecución en el peor caso de EuclidGCD(x, y), como función de x e y? (Asume que calcular x − y toma tiempo O(log x + log y).)

(c) Demuestra que el siguiente algoritmo también calcula gcd(x, y):

FastEuclidGCD(x, y):

if y = 0

return x

else if x > y

return FastEuclidGCD(y, x mod y)

else

return FastEuclidGCD(x, y mod x)

(d) ¿Cuál es el tiempo de ejecución en el peor caso de FastEuclidGCD(x, y), como función de x e y? (Asume que calcular x mod y toma tiempo O(log x · log y).)

(e) Demuestra que el siguiente algoritmo también calcula gcd(x, y):

BinaryGCD(x, y):

if x = y

return x

else if x and y are both even

return 2 · BinaryGCD(x/2, y/2)

else if x is even

return BinaryGCD(x/2, y)

else if y is even

return BinaryGCD(x, y/2)

else if x > y

return BinaryGCD((x − y)/2, y)

else

return BinaryGCD(x, (y − x)/2)

(f) ¿Cuál es el tiempo de ejecución en el peor caso de BinaryGCD(x, y), como función de x e y? (Asume que calcular x − y toma tiempo O(log x + log y), y calcular z/2 requiere tiempo O(log z).)

**Arreglos**

1. Supón que te dan un tablero de ajedrez de 2ⁿ × 2ⁿ con un cuadrado (elegido arbitrariamente) removido. Describe y analiza un algoritmo para calcular un mosaico del tablero sin huecos o superposiciones por fichas en forma de L, cada una compuesta de 3 cuadrados. Tu entrada es el entero n y dos enteros de n bits representando la fila y columna del cuadrado faltante. La salida es una lista de las posiciones y orientaciones de (4ⁿ − 1)/3 fichas. Tu algoritmo debería ejecutarse en tiempo O(4ⁿ). [Pista: Primero demuestra que tal mosaico siempre existe.]
2. Eres un visitante en una convención política (o tal vez una reunión de facultad) con n delegados; cada delegado es miembro de exactamente un partido político. Es imposible decir a qué partido político pertenece cualquier delegado; en particular, serás sumariamente expulsado de la convención si preguntas. Sin embargo, puedes determinar si cualquier par de delegados pertenece al mismo partido presentándolos entre sí. Los miembros del mismo partido político siempre se saludan con sonrisas y apretones de manos amistosos; los miembros de diferentes partidos siempre se saludan con miradas furiosas e insultos¹⁹.

(a) Supón que más de la mitad de los delegados pertenecen al mismo partido político. Describe un algoritmo eficiente que identifique a todos los miembros de este partido mayoritario.

(b) Ahora supón que hay más de dos partidos, pero un partido tiene una pluralidad: más personas pertenecen a ese partido que a cualquier otro partido. Presenta un procedimiento práctico para elegir precisamente a las personas del partido de pluralidad tan parsimoniosa­mente como sea posible, presumiendo que el partido de pluralidad está compuesto de al menos p personas. Por favor.

1. La Isla Smullyan tiene tres tipos de habitantes: los caballeros siempre dicen la verdad; los bribones siempre mienten; y los normales a veces dicen la verdad y a veces no. Todos en la isla conocen el nombre y tipo de todos los demás (caballero, bribón o normal). Quieres aprender el tipo de cada habitante.

Puedes pedirle a cualquier habitante que te diga el tipo de cualquier otro habitante. Específicamente, si preguntas "Oye X, ¿cuál es el tipo de Y?" entonces X responderá como sigue:

• Si X es un caballero, entonces X responderá con el tipo correcto de Y. • Si X es un bribón, entonces X podría responder con cualquiera de los tipos que Y no es. • Si X es un normal, entonces X podría responder con cualquiera de los tres tipos.

Los habitantes ignorarán cualquier pregunta que no sea de esta forma precisa; en particular, no puedes preguntarle a un habitante sobre su propio tipo. Hacer la misma pregunta al mismo habitante múltiples veces siempre produce la misma respuesta, así que no tiene sentido hacer cualquier pregunta más de una vez.

(a) Supón que sabes que una mayoría estricta de habitantes son caballeros. Describe un algoritmo eficiente para identificar el tipo de cada habitante.

(b) Demuestra que si a lo sumo la mitad de los habitantes son caballeros, es imposible determinar el tipo de cada habitante.

1. La mayoría del hardware gráfico incluye soporte para una operación de bajo nivel llamada blit, o transferencia de bloque, que copia rápidamente un fragmento rectangular de un mapa de píxeles (un arreglo bidimensional de valores de píxel) de una ubicación a otra. Esta es una versión bidimensional de la función estándar de biblioteca C memcpy().

Supón que queremos rotar un mapa de píxeles de n × n 90° en sentido horario. Una forma de hacer esto, al menos cuando n es una potencia de dos, es dividir el mapa de píxeles en cuatro bloques de n/2 × n/2, mover cada bloque a su posición apropiada usando una secuencia de cinco blits, y luego rotar recursivamente cada bloque. (¿Por qué cinco? Por la misma razón que el rompecabezas de la Torre de Hanoi necesita una tercera clavija.) Alternativamente, podríamos primero rotar recursivamente los bloques y luego blitearlos en su lugar.

**Figura 1.24.** Dos algoritmos para rotar un mapa de píxeles.

(a) Demuestra que ambas versiones del algoritmo son correctas cuando n es una potencia de 2.

(b) ¿Exactamente cuántos blits realiza el algoritmo cuando n es una potencia de 2?

(c) Describe cómo modificar el algoritmo para que funcione para n arbitrario, no solo potencias de 2. ¿Cuántos blits realiza tu algoritmo modificado?

(d) ¿Cuál es el tiempo de ejecución de tu algoritmo si un blit de k × k toma tiempo O(k²)?

(e) ¿Qué pasa si un blit de k × k toma solo tiempo O(k)?

1. Un arreglo A[0 .. n − 1] de n números distintos es bitónico si hay índices únicos i y j tales que A[(i − 1) mod n] < A[i] > A[(i + 1) mod n] y A[(j − 1) mod n] > A[j] < A[(j + 1) mod n]. En otras palabras, una secuencia bitónica consiste de una secuencia creciente seguida por una secuencia decreciente, o puede desplazarse circularmente para convertirse en tal. Por ejemplo,

4 6 9 8 7 5 1 2 3 es bitónica, pero  
3 6 9 8 7 5 1 2 4 no es bitónica.

Describe y analiza un algoritmo para encontrar el elemento más pequeño en un arreglo bitónico de n elementos en tiempo O(log n). Puedes asumir que los números en el arreglo de entrada son distintos.

1. Supón que nos dan un arreglo A[1 .. n] de n enteros distintos, que podrían ser positivos, negativos o cero, ordenados en orden creciente para que A[1] < A[2] < ··· < A[n].

(a) Describe un algoritmo rápido que calcule un índice i tal que A[i] = i o reporte correctamente que no existe tal índice.

(b) Supón que sabemos de antemano que A[1] > 0. Describe un algoritmo aún más rápido que calcule un índice i tal que A[i] = i o reporte correctamente que no existe tal índice. [Pista: Esto es realmente fácil.]

1. Supón que nos dan un arreglo A[1 .. n] con la propiedad especial de que A[1] ≥ A[2] y A[n − 1] ≤ A[n]. Decimos que un elemento A[x] es un mínimo local si es menor o igual que ambos sus vecinos, o más formalmente, si A[x − 1] ≥ A[x] y A[x] ≤ A[x + 1]. Por ejemplo, hay seis mínimos locales en el siguiente arreglo:

9 ↓ 7 7 2 ↓ 1 3 7 5 ↓ 4 7 ↓ 3 ↓ 3 4 8 ↓ 6 9

Obviamente podemos encontrar un mínimo local en tiempo O(n) escaneando a través del arreglo. Describe y analiza un algoritmo que encuentre un mínimo local en tiempo O(log n). [Pista: Con las condiciones de frontera dadas, el arreglo debe tener al menos un mínimo local. ¿Por qué?]

1. Supón que te dan un arreglo ordenado de n números distintos que ha sido rotado k pasos, para algún entero desconocido k entre 1 y n − 1. Es decir, se te da un arreglo A[1 .. n] tal que algún prefijo A[1 .. k] está ordenado en orden creciente, el sufijo correspondiente A[k + 1 .. n] está ordenado en orden creciente, y A[n] < A[1].

Por ejemplo, podrías recibir el siguiente arreglo de 16 elementos (donde k = 10):

9 13 16 18 19 23 28 31 37 42 1 3 4 5 7 8

(a) Describe y analiza un algoritmo para calcular el entero desconocido k.

(b) Describe y analiza un algoritmo para determinar si el arreglo dado contiene un número dado x.

1. Al final del segundo acto del blockbuster de acción Fast and Impossible XIII¾: The Last Guardians of Expendable Justice Reloaded, el villano Dr. Metaphor hipnotiza a toda la Liga/Fuerza/Escuadrón de Héroes, los organiza en una línea larga al borde de un acantilado, e instruye a cada héroe a disparar a los héroes más altos más cercanos a su izquierda y derecha, a una señal preestablecida.

Supón que nos dan las alturas de todos los n héroes, en orden de izquierda a derecha, en un arreglo Ht[1 .. n]. (Para evitar argumentos salariales, los productores insistieron en que no dos héroes tienen la misma altura.) Entonces podemos calcular los objetivos Izquierdo y Derecho de cada héroe en tiempo O(n²) usando el siguiente algoritmo de fuerza bruta.

WhoTargetsWhom(Ht[1 .. n]):

for j ← 1 to n

《《Encontrar el objetivo izquierdo L[j] para el héroe j》》

L[j] ← None

for i ← 1 to j − 1

if Ht[i] > Ht[j]

L[j] ← i

《《Encontrar el objetivo derecho R[j] para el héroe j》》

R[j] ← None

for k ← n down to j + 1

if Ht[k] > Ht[j]

R[j] ← k

return L[1 .. n], R[1 .. n]

(a) Describe un algoritmo de divide y vencerás que calcule la salida de WhoTargetsWhom en tiempo O(n log n).

(b) Demuestra que al menos ⌊n/2⌋ de los n héroes son objetivos. Es decir, demuestra que los arreglos de salida R[0 .. n − 1] y L[0 .. n − 1] contienen al menos ⌊n/2⌋ valores distintos (distintos de None).

(c) ¡Ay, el plan diabólico del Dr. Metaphor es exitoso! A la señal preestablecida, todos los héroes disparan simultáneamente a sus objetivos, y todos los objetivos caen por el acantilado, aparentemente muertos. Metaphor repite su experimento diabólico una y otra vez; después de cada masacre, obliga a los héroes restantes a elegir nuevos objetivos, siguiendo el mismo algoritmo, y luego disparar a sus objetivos en la siguiente señal. Eventualmente, solo el miembro más bajo del Equipo/Alianza/Grupo de Héroes queda vivo²⁰.

Describe y analiza un algoritmo para calcular el número de rondas antes de que el proceso mortal del Dr. Metaphor finalmente termine. Para crédito completo, tu algoritmo debería ejecutarse en tiempo O(n).

1. Eres un concursante en el exitoso programa de juegos "¡Vence a Tus Vecinos!" Se te presenta una cuadrícula de m × n de cajas, cada una conteniendo un número único. Cuesta $100 abrir una caja. Tu objetivo es encontrar una caja cuyo número sea más grande que sus vecinos en la cuadrícula (arriba, abajo, izquierda y derecha). Si gastas menos dinero que cualquiera de tus oponentes, ganas un viaje de una semana para dos a Las Vegas y un suministro de un año de Rice-A-Ronti™, al cual eres irremediablemente adicto.

(a) Supón m = 1. Describe un algoritmo que encuentre un número que sea más grande que cualquiera de sus vecinos. ¿Cuántas cajas abre tu algoritmo en el peor caso?

♥(b) Supón m = n. Describe un algoritmo que encuentre un número que sea más grande que cualquiera de sus vecinos. ¿Cuántas cajas abre tu algoritmo en el peor caso?

♣♥(c) Demuestra que tu solución a la parte (b) es óptima hasta un factor constante.

1. (a) Sea n = 2ℓ − 1 para algún entero positivo ℓ. Supón que alguien afirma tener un arreglo no ordenado A[1 .. n] de cadenas de ℓ bits distintas; así, exactamente una cadena de ℓ bits no aparece en A. Supón además que la única forma en que podemos acceder a A es llamando a la función FetchBit(i, j), que devuelve el j-ésimo bit de la cadena A[i] en tiempo O(1). Describe un algoritmo para encontrar la cadena faltante en A usando solo O(n) llamadas a FetchBit.

♥(b) Ahora supón n = 2ℓ − k para algunos enteros positivos k y ℓ, y nuevamente se nos da un arreglo A[1 .. n] de cadenas de ℓ bits distintas. Describe un algoritmo para encontrar las k cadenas que faltan de A usando solo O(n log k) llamadas a FetchBit.

**Árboles**

1. Para este problema, un subárbol de un árbol binario significa cualquier subgrafo conectado. Un árbol binario es completo si cada nodo interno tiene dos hijos, y cada hoja tiene exactamente la misma profundidad. Describe y analiza un algoritmo recursivo para calcular el subárbol completo más grande de un árbol binario dado. Tu algoritmo debería devolver tanto la raíz como la profundidad de este subárbol. Ver Figura 1.26 para un ejemplo.

**Figura 1.26.** El subárbol completo más grande de este árbol binario tiene profundidad 3.

1. Sea T un árbol binario con n vértices. Eliminar cualquier vértice v divide T en a lo sumo tres subárboles, conteniendo el hijo izquierdo de v (si existe), el hijo derecho de v (si existe), y el padre de v (si existe). Llamamos a v un vértice central si cada uno de estos árboles más pequeños tiene a lo sumo n/2 vértices. Ver Figura 1.27 para un ejemplo.

Describe y analiza un algoritmo para encontrar un vértice central en un árbol binario arbitrario dado. [Pista: Primero demuestra que cada árbol tiene un vértice central.]

**Figura 1.27.** Eliminando un vértice central en un árbol binario de 34 nodos, dejando subárboles con 14, 7 y 12 nodos.

1. (a) El Profesor George O'Jungle tiene un árbol binario de 27 nodos, en el cual cada nodo está etiquetado con una letra única del alfabeto romano o el carácter &. Los recorridos en preorden y postorden del árbol visitan los nodos en el siguiente orden:

• Preorden: I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X • Postorden: H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I

Dibuja el árbol binario de George.

(b) Recuerda que un árbol binario es lleno si cada nodo no hoja tiene exactamente dos hijos.

i. Describe y analiza un algoritmo recursivo para reconstruir un árbol binario lleno arbitrario, dadas sus secuencias de nodos en preorden y postorden como entrada.

ii. Demuestra que no hay algoritmo para reconstruir un árbol binario arbitrario de sus secuencias de nodos en preorden y postorden.

(c) Describe y analiza un algoritmo recursivo para reconstruir un árbol binario arbitrario, dadas sus secuencias de nodos en preorden e inorden como entrada.

(d) Describe y analiza un algoritmo recursivo para reconstruir un árbol binario de búsqueda arbitrario, dada solo su secuencia de nodos en preorden.

♥(e) Describe y analiza un algoritmo recursivo para reconstruir un árbol binario de búsqueda arbitrario, dada solo su secuencia de nodos en preorden, en tiempo O(n).

En las partes (b)–(e), asume que todas las claves son distintas y que la entrada es consistente con al menos un árbol binario.

1. Supón que tenemos n puntos esparcidos dentro de una caja bidimensional. Un kd-tree²¹ subdivide recursivamente los puntos como sigue. Si la caja no contiene puntos en su interior, hemos terminado. De lo contrario, dividimos la caja en dos cajas más pequeñas con una línea vertical, a través de un punto mediano dentro de la caja (no en su frontera), particionando los puntos tan uniformemente como sea posible. Luego construimos recursivamente un kd-tree para los puntos en cada una de las dos cajas más pequeñas, después de rotarlas 90 grados. Así, alternamos entre dividir verticalmente y dividir horizontalmente en cada nivel de recursión. Las cajas vacías finales se llaman celdas.

**Figura 1.28.** Un kd-tree para 15 puntos. La línea punteada cruza las cuatro celdas sombreadas.

(a) ¿Cuántas celdas hay, como función de n? Demuestra que tu respuesta es correcta.

(b) En el peor caso, ¿exactamente cuántas celdas puede cruzar una línea horizontal, como función de n? Demuestra que tu respuesta es correcta. Asume que n = 2k − 1 para algún entero k. [Pista: Hay más de una función f tal que f(16) = 4.]

(c) Supón que nos dan n puntos almacenados en un kd-tree. Describe y analiza un algoritmo que cuente el número de puntos arriba de una línea horizontal (como la línea punteada en la figura) tan rápido como sea posible. [Pista: Usa la parte (b).]

(d) Describe y analiza un algoritmo eficiente que cuente, dado un kd-tree conteniendo n puntos, el número de puntos que se encuentran dentro de un rectángulo R con lados horizontales y verticales. [Pista: Usa la parte (c).]

♥41. Bob Ratenbur, un nuevo estudiante en CS 225, está tratando de escribir código para realizar recorridos en preorden, inorden y postorden de árboles binarios. Bob más o menos entiende la idea básica detrás de los algoritmos de recorrido, pero cada vez que realmente trata de implementarlos, sigue confundiendo las llamadas recursivas. Cinco minutos antes de la fecha límite, Bob frenéticamente envía código con la siguiente estructura:

PreOrder(v):

if v = Null

return

else

print label(v)

Order(left(v))

Order(right(v))

InOrder(v):

if v = Null

return

else

Order(left(v))

print label(v)

Order(right(v))

PostOrder(v):

if v = Null

return

else

Order(left(v))

Order(right(v))

print label(v)

Cada en este pseudocódigo oculta uno de los prefijos Pre, In o Post. Además, cada una de las siguientes llamadas de función aparece exactamente una vez en el código enviado por Bob:

PreOrder(left(v)) PreOrder(right(v))  
InOrder(left(v)) InOrder(right(v))  
PostOrder(left(v)) PostOrder(right(v))

Así, hay precisamente 36 posibilidades para el código de Bob. Desafortunadamente, Bob accidentalmente eliminó su código fuente después de enviar el ejecutable, así que ni tú ni él saben qué funciones fueron llamadas dónde.

Ahora supón que se te da la salida de los algoritmos de recorrido de Bob, ejecutados en algún árbol binario desconocido T. La salida de Bob ha sido útilmente analizada en tres arreglos Pre[1 .. n], In[1 .. n] y Post[1 .. n]. Puedes asumir que estas secuencias de recorrido son consistentes con exactamente un árbol binario T; en particular, las etiquetas de vértice del árbol desconocido T son distintas, y cada nodo interno en T tiene exactamente dos hijos.

(a) Describe un algoritmo para reconstruir el árbol desconocido T de las secuencias de recorrido dadas.

(b) Describe un algoritmo que reconstruya el código de Bob de las secuencias de recorrido dadas, o reporte correctamente que las secuencias de recorrido son consistentes con más de un conjunto de algoritmos.

Por ejemplo, dada la entrada

Pre[1 .. n] = [H A E C B I F G D]  
In[1 .. n] = [A H D C E I F B G]  
Post[1 .. n] = [A E I B F C D G H]

tu primer algoritmo debería devolver el siguiente árbol:

H

/ \

A D

/ \

C G

/ /

E B

/ /

I F

y tu segundo algoritmo debería reconstruir el siguiente código:

PreOrder(v):

if v = Null

return

else

print label(v)

PreOrder(left(v))

PostOrder(right(v))

InOrder(v):

if v = Null

return

else

PostOrder(left(v))

print label(v)

PreOrder(right(v))

PostOrder(v):

if v = Null

return

else

InOrder(left(v))

InOrder(right(v))

print label(v)

♥42. Sea T un árbol binario cuyos nodos almacenan valores numéricos distintos. Recuerda que T es un árbol binario de búsqueda si y solo si (1) T está vacío, o (2) T satisface las siguientes condiciones recursivas:

• El subárbol izquierdo de T es un árbol binario de búsqueda. • Todos los valores en el subárbol izquierdo son menores que el valor en la raíz. • El subárbol derecho de T es un árbol binario de búsqueda. • Todos los valores en el subárbol derecho son mayores que el valor en la raíz.

Considera el siguiente par de operaciones en árboles binarios:

• Rotar un nodo arbitrario hacia arriba²².

y x

/ \ / \

x C → A y

/ \ / \

A B B C

• Intercambiar los subárboles izquierdo y derecho de un nodo arbitrario.

x x

/ \ → / \

A B B A

En ambas operaciones, algunos, todos o ninguno de los subárboles A, B y C podrían estar vacíos.

(a) Describe un algoritmo para transformar un árbol binario arbitrario de n nodos con valores de nodo distintos en un árbol binario de búsqueda, usando a lo sumo O(n²) rotaciones e intercambios. La Figura 1.29 muestra una secuencia de ocho operaciones que transforma un árbol binario de cinco nodos en un árbol binario de búsqueda.

**Figura 1.29.** "Ordenando" un árbol binario: rotar 2, rotar 2, intercambiar 3, rotar 3, rotar 4, intercambiar 3, rotar 2, intercambiar 4.

Tu algoritmo no puede modificar directamente punteros padre o hijo, crear nuevos nodos o eliminar nodos viejos; la única forma de modificar el árbol es a través de rotaciones e intercambios.

Por otro lado, puedes calcular cualquier cosa que quieras de forma gratuita, siempre que ese cálculo no modifique el árbol; el tiempo de ejecución de tu algoritmo se define como el número de rotaciones e intercambios que realiza.

♥(b) Describe un algoritmo para transformar un árbol binario arbitrario de n nodos en un árbol binario de búsqueda, usando a lo sumo O(n log n) rotaciones e intercambios.

(c) Demuestra que cualquier árbol binario de búsqueda de n nodos puede transformarse en cualquier otro árbol binario de búsqueda con los mismos valores de nodo, usando solo O(n) rotaciones (y ningún intercambio).

♥(d) Problema abierto: Describe un algoritmo para transformar un árbol binario arbitrario de n nodos en un árbol binario de búsqueda usando solo O(n) rotaciones e intercambios, o demuestra que no es posible tal algoritmo. [Pista: No creo que sea posible.]

**Notas al pie:**

¹⁰Mi presentación simplifica la historia real ligeramente. De hecho, Karatsuba propuso un algoritmo basado en la fórmula (a + b)(c + d) − ac − bd = bc + ad. Este algoritmo también se ejecuta en tiempo O(n^lg 3), pero la recurrencia real es ligeramente más desordenada: a − b y c − d siguen siendo números de m dígitos, pero a + b y c + d podrían cada uno tener m + 1 dígitos. La simplificación presentada aquí se debe a Donald Knuth.

¹¹Ver http://algorithms.wtf para notas sobre transformadas rápidas de Fourier.

¹²Schönhage-Strassen es en realidad el algoritmo más rápido en la práctica para multiplicar enteros con más de aproximadamente 75000 dígitos; los algoritmos más recientes de Fürer, Harvey, van der Hoeven y otros serían más rápidos "en la práctica" solo para enteros con más dígitos que partículas hay en el universo.

¹³De Viribus Quantitatis [Sobre los Poderes de los Números] es una obra temprana importante sobre matemáticas recreacionales y tal vez el tratado más antiguo que sobrevive sobre magia. Pacioli es mejor conocido por Summa de Arítmetica, una enciclopedia casi completa de matemáticas de finales del siglo XV, que incluyó la primera descripción de contabilidad por partida doble.

¹⁴Partes de esta historia son realmente verdaderas.

¹⁵El número exacto óptimo en el peor caso de volteos requeridos para ordenar n panqueques (ya sean quemados o no quemados) es un problema abierto de larga data; solo haz lo mejor que puedas.

¹⁶La mediana de cuatro elementos es el segundo más pequeño o el segundo más grande. En 2014, Ke Chen y Adrian Dumitrescu demostraron que si modificamos Mom4Select para encontrar elementos segundo-más-pequeños cuando k < n/2 y elementos segundo-más-grandes cuando k > n/2, ¡el algoritmo resultante se ejecuta en tiempo O(n)! Ver su artículo "Select with Groups of 3 or 4 Takes Linear Time" (WADS 2015, arXiv:1409.3600) para detalles.

¹⁷El algoritmo de Euclides a veces se describe incorrectamente como el algoritmo recursivo más antiguo, o incluso el algoritmo no trivial más antiguo, aunque el algoritmo egipcio de duplación y mediación —que es tanto no trivial como recursivo— es anterior a Euclides por al menos 1500 años.

¹⁸Euclides no hizo esto. La Proposición 1 en Elementos Libro VII establece que si EuclidGCD(x, y) = 1, entonces x e y son relativamente primos (es decir, gcd(x, y) = 1), pero la prueba solo considera el caso especial x mod (y mod (x mod y)) = 1. La Proposición 2 establece que si x e y no son relativamente primos, entonces EuclidGCD(x, y) = gcd(x, y), pero la prueba solo considera los casos especiales gcd(x, y) = y y gcd(x, y) = y mod (x mod y). Finalmente, estas dos Proposiciones no constituyen una prueba completa de que EuclidGCD es correcto. No seas como Euclides.

¹⁹La política del mundo real es mucho más desordenada que este modelo simplificado, ¡pero este es un libro de teoría!

²⁰En el emocionante acto final, Retcon the Squirrel, el último miembro sobreviviente del Equipo/Grupo/Sociedad de Héroes, salva a todos viajando al pasado y reemplazando retroactivamente a los otros n − 1 héroes con esculturas de globos realistas. Así que, sí, básicamente es Avengers: Endgame.

²¹El término "kd-tree" (pronunciado "kay dee tree") fue originalmente una abreviatura de "árbol k-dimensional", pero el uso moderno ignora esta etimología, en parte porque nadie en su sano juicio usaría la letra k para denotar dimensión en lugar de la obviamente superior d. La consistencia etimológica requeriría llamar a la estructura de datos en este problema un "2d-tree" (o tal vez un "árbol 2-d"), pero la nomenclatura estándar ahora es "kd-tree bidimensional". Ver también: B-tree (tal vez), forma alfa, esqueleto beta, red épsilon, Río Potomac, Río Mississippi, Lago Michigan, Lago Tahoe, Isla Manhattan, La Brea Tar Pits, Desierto del Sahara, Monte Kilimanjaro, Vietnam del Sur, Timor Oriental, la Galaxia Vía Láctea, la Ciudad de Townsville, y automóviles que se conducen solos.

²²Las rotaciones preservan la secuencia inorden de nodos en un árbol binario. Parcialmente por esta razón, las rotaciones se usan para mantener varios tipos de árboles binarios de búsqueda balanceados, incluyendo árboles AVL, árboles rojo-negro, árboles splay, árboles chivo expiatorio y treaps. Ver http://algorithms.wtf para notas sobre la mayoría de estas estructuras de datos.

---------------------------------------------------------------------------------------------------------------------

Donde, sin embargo, la ambigüedad no puede aclararse, ya sea por la regla de la fe o por el contexto, no hay nada que nos impida puntuar la oración según cualquier método que elijamos de aquellos que se sugieren.

— Agustín de Hipona, De doctrina Christiana (397 d.C.)

Traducido por Marcus Dods (1892)

Dejé caer mi cena, y corrí de vuelta al laboratorio. Allí, en mi emoción, probé el contenido de cada vaso y plato de evaporación en la mesa. Afortunadamente para mí, ninguno contenía ningún líquido corrosivo o venenoso.

— Constantine Fahlberg sobre su descubrimiento de la sacarina,

Scientific American (1886)

El mayor desafío para cualquier pensador es plantear el problema de una manera que permita una solución.

— atribuido a Bertrand Russell

Cuando llegues a una bifurcación en el camino, tómala.

— Yogi Berra (dando direcciones a su casa)

**2 Backtracking**

Este capítulo describe otra estrategia recursiva importante llamada **backtracking** (retroceso).

Un algoritmo de backtracking trata de construir una solución a un problema computacional de manera incremental, una pequeña pieza a la vez. Cada vez que el algoritmo necesita decidir entre múltiples alternativas para el siguiente componente de la solución, evalúa recursivamente cada alternativa y luego elige la mejor.

**2.1 N Reinas**

El problema prototípico de backtracking es el clásico **Problema de las n Reinas**, propuesto por primera vez por el entusiasta alemán del ajedrez Max Bezzel en 1848 (bajo su pseudónimo "Schachfreund") para el tablero estándar de 8 × 8 y por François-Joseph Eustache Lionnet en 1869 para el tablero más general de n × n. El problema consiste en colocar n reinas en un tablero de ajedrez de n × n, de tal manera que ninguna de las dos reinas se ataque entre sí.

Para lectores no familiarizados con las reglas del ajedrez, esto significa que no hay dos reinas en la misma fila, la misma columna, o la misma diagonal.

**Figura 2.1.** Primera solución de Gauss al problema de las 8 reinas, representada por el arreglo [5, 7, 1, 4, 2, 8, 6, 3]

En una carta escrita a su amigo Heinrich Schumacher en 1850, el eminente matemático Carl Friedrich Gauss escribió que uno podía confirmar fácilmente la afirmación de Franz Nauck de que el problema de las Ocho Reinas tiene 92 soluciones mediante ensayo y error en unas pocas horas. ("Schwer ist es übrigens nicht, durch ein methodisches Tatonniren sich diese Gewissheit zu verschaffen, wenn man 1 oder ein paar Stunden daran wenden will.") Su descripción Tatonniren proviene del francés tâtonner, que significa sentir, tantear, o buscar a ciegas, como si estuviera en la oscuridad.

La carta de Gauss describía la siguiente estrategia recursiva para resolver el problema de las n-reinas; la misma estrategia fue descrita en 1882 por el matemático recreacional francés Édouard Lucas, quien atribuyó el método a Emmanuel Laquière. Colocamos reinas en el tablero una fila a la vez, comenzando con la fila superior. Para colocar la r-ésima reina, probamos metódicamente todos los n cuadrados en la fila r de izquierda a derecha en un simple bucle for. Si un cuadrado particular está atacado por una reina anterior, ignoramos ese cuadrado; de lo contrario, colocamos tentativamente una reina en ese cuadrado y buscamos recursivamente colocaciones consistentes de las reinas en filas posteriores.

La Figura 2.2 muestra el algoritmo resultante, que enumera recursivamente todas las soluciones completas de n-reinas que son consistentes con una solución parcial dada. Siguiendo a Gauss, representamos las posiciones de las reinas usando un arreglo Q[1 .. n], donde Q[i] indica qué cuadrado en la fila i contiene una reina. Cuando se llama a PlaceQueens, el parámetro de entrada r es el índice de la primera fila vacía, y el prefijo Q[1 ..r − 1] contiene las posiciones de las primeras r − 1 reinas. En particular, para calcular todas las soluciones de n-reinas sin restricciones, llamaríamos PlaceQueens(Q[1 .. n], 1). El bucle for externo considera todas las colocaciones posibles de una reina en la fila r; el bucle for interno verifica si una colocación candidata en la fila r es consistente con las reinas que ya están en las primeras r − 1 filas.

La ejecución de PlaceQueens puede ilustrarse usando un árbol de recursión. Cada nodo en este árbol corresponde a un subproblema recursivo, y por tanto a una solución parcial legal; en particular, la raíz corresponde al tablero vacío

**2.1. N Reinas**

PlaceQueens(Q[1 .. n],r):

if r = n + 1

print Q[1 .. n]

else

for j ← 1 to n

legal ← True

for i ← 1 to r − 1

if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)

legal ← False

if legal

Q[r] ← j

PlaceQueens(Q[1 .. n],r + 1) 〈〈¡Recursión!〉〉

**Figura 2.2.** Algoritmo de retroceso de Gauss y Laquière para el problema de las n reinas.

(con r = 0). Las aristas en el árbol de recursión corresponden a llamadas recursivas. Las hojas corresponden a soluciones parciales que no pueden extenderse más, ya sea porque ya hay una reina en cada fila, o porque cada posición en la siguiente fila vacía está atacada por una reina existente. La búsqueda de retroceso para soluciones completas es equivalente a una búsqueda en profundidad de este árbol.

**Figura 2.3.** El árbol de recursión completo del algoritmo de Gauss y Laquière para el problema de las 4 reinas.

PlaceQueens(Q[1 .. n], r):

if r = n + 1

print Q[1 .. n]

else

for j ← 1 to n

legal ← True

for i ← 1 to r − 1

if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)

legal ← False

if legal

Q[r] ← j

PlaceQueens(Q[1 .. n], r + 1) 〈〈¡Recursión!〉〉

**Figura 2.2.** Algoritmo de retroceso de Gauss y Laquière para el problema de las n reinas.

(con r = 0). Las aristas en el árbol de recursión corresponden a llamadas recursivas. Las hojas corresponden a soluciones parciales que no pueden extenderse más, ya sea porque ya hay una reina en cada fila, o porque cada posición en la siguiente fila vacía está atacada por una reina existente. La búsqueda de retroceso para soluciones completas es equivalente a una búsqueda en profundidad de este árbol.

**Figura 2.3.** El árbol de recursión completo del algoritmo de Gauss y Laquière para el problema de las 4 reinas.

**2.2 Árboles de Juego**

Considere el siguiente juego simple de dos jugadores¹ jugado en una cuadrícula cuadrada de n × n con un borde de cuadrados; llamemos a los jugadores Horace Fahlberg-Remsen y Vera Rebaudi.² Cada jugador tiene n fichas que mueven a través del tablero de un lado al otro. Las fichas de Horace comienzan en el borde izquierdo, una en cada fila, y se mueven horizontalmente hacia la derecha; simétricamente, las fichas de Vera comienzan en el borde superior, una en cada columna, y se mueven verticalmente hacia abajo. Los jugadores alternan turnos. En cada uno de sus turnos, Horace mueve una de sus fichas un paso a la derecha hacia un cuadrado vacío, o salta una de sus fichas sobre exactamente una de las fichas de Vera hacia un cuadrado vacío dos pasos a la derecha. Si no hay movimientos o saltos legales disponibles, Horace simplemente pasa. Similarmente, Vera mueve o salta una de sus fichas hacia abajo en cada uno de sus turnos, a menos que no sean posibles movimientos o saltos. El primer jugador en mover todas sus fichas fuera del borde del tablero gana. (No es difícil probar que mientras haya fichas en el tablero, al menos un jugador puede moverse legalmente, y por lo tanto alguien eventualmente gana.)

**Figura 2.4.** Vera gana el juego de sobres de azúcar falsa de 3 × 3.

¹ No sé cómo se llama este juego, o incluso si estoy reportando las reglas correctamente; lo aprendí (o algo parecido) de Lenny Pitt, quien recomendó jugarlo con sobres de azúcar falsa en restaurantes.

² Constantin Fahlberg e Ira Remsen sintetizaron sacarina por primera vez en 1878, mientras Fahlberg era un postdoctorado en el laboratorio de Remsen investigando derivados del alquitrán de hulla. En 1900, Ovidio Rebaudi publicó el primer análisis químico de ka'a he'ê, una planta medicinal cultivada por los Guaraní durante más de 1500 años, ahora más comúnmente conocida como Stevia rebaudiana.

A menos que hayas visto este juego antes³, probablemente no tienes idea de cómo jugarlo bien. Sin embargo, hay un algoritmo de retroceso relativamente simple que puede jugar este juego—o cualquier juego de dos jugadores sin aleatoriedad o información oculta que termine después de un número finito de movimientos—perfectamente. Es decir, si te dejamos caer en el medio de un juego, y es posible ganar contra otro jugador perfecto, el algoritmo te dirá cómo ganar.

Un estado del juego consiste en las ubicaciones de todas las piezas y la identidad del jugador actual. Estos estados pueden conectarse en un árbol de juego, que tiene una arista del estado x al estado y si y solo si el jugador actual en el estado x puede moverse legalmente al estado y. La raíz del árbol de juego es la posición inicial del juego, y cada camino desde la raíz hasta una hoja es un juego completo.

**Figura 2.5.** Los primeros dos niveles del árbol de juego de sobres de azúcar falsa.

Para navegar a través de este árbol de juego, definimos recursivamente que un estado de juego es bueno o malo como sigue:

• Un estado de juego es **bueno** si el jugador actual ya ha ganado, o si el jugador actual puede moverse a un estado malo para el jugador oponente.

• Un estado de juego es **malo** si el jugador actual ya ha perdido, o si cada movimiento disponible lleva a un estado bueno para el jugador oponente.

Equivalentemente, un nodo no-hoja en el árbol de juego es bueno si tiene al menos un hijo malo, y un nodo no-hoja es malo si todos sus hijos son buenos. Por inducción, cualquier jugador que encuentre el juego en un estado bueno en su turno puede ganar el juego, incluso si su oponente juega perfectamente; por otro lado, comenzando desde un estado malo, un jugador puede ganar solo si su oponente comete un error. Esta definición recursiva fue propuesta por Ernst Zermelo en 1913.⁴

³ Si lo has hecho, ¡por favor dime dónde! ⁴ De hecho, Zermelo consideró la clase más sutil de juegos que tienen un número finito de estados, pero que permiten secuencias infinitas de movimientos. (Zermelo definió el juego infinito como un empate.)

A menos que hayas visto este juego antes³, probablemente no tienes idea de cómo jugarlo bien. Sin embargo, hay un algoritmo de retroceso relativamente simple que puede jugar este juego—o cualquier juego de dos jugadores sin aleatoriedad o información oculta que termine después de un número finito de movimientos—perfectamente. Es decir, si te dejamos caer en el medio de un juego, y es posible ganar contra otro jugador perfecto, el algoritmo te dirá cómo ganar.

Un estado del juego consiste en las ubicaciones de todas las piezas y la identidad del jugador actual. Estos estados pueden conectarse en un árbol de juego, que tiene una arista del estado x al estado y si y solo si el jugador actual en el estado x puede moverse legalmente al estado y. La raíz del árbol de juego es la posición inicial del juego, y cada camino desde la raíz hasta una hoja es un juego completo.

**Figura 2.5.** Los primeros dos niveles del árbol de juego de sobres de azúcar falsa.

Para navegar a través de este árbol de juego, definimos recursivamente que un estado de juego es bueno o malo como sigue:

• Un estado de juego es **bueno** si el jugador actual ya ha ganado, o si el jugador actual puede moverse a un estado malo para el jugador oponente.

• Un estado de juego es **malo** si el jugador actual ya ha perdido, o si cada movimiento disponible lleva a un estado bueno para el jugador oponente.

Equivalentemente, un nodo no-hoja en el árbol de juego es bueno si tiene al menos un hijo malo, y un nodo no-hoja es malo si todos sus hijos son buenos. Por inducción, cualquier jugador que encuentre el juego en un estado bueno en su turno puede ganar el juego, incluso si su oponente juega perfectamente; por otro lado, comenzando desde un estado malo, un jugador puede ganar solo si su oponente comete un error. Esta definición recursiva fue propuesta por Ernst Zermelo en 1913.⁴

Esta definición recursiva sugiere inmediatamente el siguiente algoritmo de retroceso recursivo para determinar si un estado de juego dado es bueno o malo. En su núcleo, este algoritmo es solo una búsqueda en profundidad del árbol de juego; equivalentemente, ¡el árbol de juego es el árbol de recursión del algoritmo! Una modificación simple de este algoritmo de retroceso encuentra un buen movimiento (o incluso todos los posibles buenos movimientos) si la entrada es un estado de juego bueno.

PlayAnyGame(X, player):

if player has already won in state X

return Good

if player has already lost in state X

return Bad

for all legal moves X → Y

if PlayAnyGame(Y,¬player) = Bad

return Good 〈〈X → Y is a good move〉〉

return Bad 〈〈There are no good moves〉〉

Todos los programas de juego están basados en última instancia en esta estrategia simple de retroceso. Sin embargo, dado que la mayoría de los juegos tienen un número enorme de estados, no es posible atravesar todo el árbol de juego en la práctica. En su lugar, los programas de juego emplean otras heurísticas⁵ para podar el árbol de juego, ignorando estados que son obviamente (o "obviamente") buenos o malos, o al menos mejores o peores que otros estados, y/o cortando el árbol a cierta profundidad (o nivel) y usando una heurística más eficiente para evaluar las hojas.

⁵ Una heurística es un algoritmo que no funciona. (Excepto en la práctica. A veces. Tal vez.)

**2.3 Suma de Subconjuntos**

Consideremos un problema más complicado, llamado SubsetSum: Dado un conjunto X de enteros positivos y un entero objetivo T, ¿existe un subconjunto de elementos en X que sumen T? Nótese que puede haber más de un subconjunto así. Por ejemplo, si X = {8, 6, 7, 5, 3, 10, 9} y T = 15, la respuesta es Verdadero, porque los subconjuntos {8, 7} y {7, 5, 3} y {6, 9} y {5, 10} todos suman 15. Por otro lado, si X = {11, 6, 5, 1, 7, 13, 12} y T = 15, la respuesta es Falso.

Hay dos casos triviales. Si el valor objetivo T es cero, entonces podemos devolver inmediatamente Verdadero, porque el conjunto vacío es un subconjunto de cada conjunto X, y los elementos del conjunto vacío suman cero.⁶ Por otro lado, si T < 0, o si T ≠ 0 pero el conjunto X está vacío, entonces podemos devolver inmediatamente Falso.

Para el caso general, considera un elemento arbitrario x ∈ X. (Ya hemos manejado el caso donde X está vacío.) Existe un subconjunto de X que suma T si y solo si una de las siguientes declaraciones es verdadera:

⁶ ... porque ¿a qué otra cosa podrían sumar?

• Existe un subconjunto de X que incluye x y cuya suma es T. • Existe un subconjunto de X que excluye x y cuya suma es T.

En el primer caso, debe existir un subconjunto de X \ {x} que sume T − x; en el segundo caso, debe existir un subconjunto de X \ {x} que sume T. Así que podemos resolver SubsetSum(X, T) reduciéndolo a dos instancias más simples: SubsetSum(X {x}, T − x) y SubsetSum(X \ {x}, T). El algoritmo recursivo resultante se muestra abajo.

〈〈¿Algún subconjunto de X suma T?〉〉

SubsetSum(X, T):

if T = 0

return True

else if T < 0 or X = ∅

return False

else

x ← any element of X

with ← SubsetSum(X \ {x}, T − x) 〈〈¡Recursión!〉〉

wout ← SubsetSum(X \ {x}, T) 〈〈¡Recursión!〉〉

return (with ∨ wout)

**Corrección**

Probar que este algoritmo es correcto es un ejercicio directo de inducción. Si T = 0, entonces los elementos del subconjunto vacío suman T, así que Verdadero es la salida correcta. De lo contrario, si T es negativo o el conjunto X está vacío, entonces ningún subconjunto de X suma T, así que Falso es la salida correcta. De lo contrario, si existe un subconjunto que suma T, entonces o contiene X[n] o no lo contiene, y el Hada de la Recursión verifica correctamente cada una de esas posibilidades. Hecho.

**Análisis**

Para analizar el algoritmo, tenemos que describir algunos detalles de implementación más precisamente. Para empezar, asumamos que el conjunto de entrada X se da como un arreglo X[1 .. n].

El algoritmo recursivo anterior nos permite elegir cualquier elemento x ∈ X en el caso recursivo principal. Puramente por eficiencia, es útil elegir un elemento x tal que el subconjunto restante X \ {x} tenga una descripción concisa, que pueda calcularse rápidamente, de manera que configurar las llamadas recursivas requiera sobrecarga mínima. Específicamente, dejaremos que x sea el último elemento X[n]; entonces el subconjunto X \ {x} se almacena en el prefijo X[1 .. n − 1]. Pasar una copia completa de este prefijo a las llamadas recursivas tomaría demasiado tiempo—necesitamos tiempo Θ(n) solo para hacer la copia—así que en su lugar, pasamos solo dos valores: una referencia al arreglo (o su dirección de inicio) y la longitud del prefijo. (Alternativamente, podríamos evitar pasar una referencia a X a cada llamada recursiva haciendo X una variable global.)

〈〈¿Algún subconjunto de X[1 .. i] suma T?〉〉

SubsetSum(X, i, T):

if T = 0

return True

else if T < 0 or i = 0

return False

else

with ← SubsetSum(X, i − 1, T − X[i]) 〈〈¡Recursión!〉〉

wout ← SubsetSum(X, i − 1, T) 〈〈¡Recursión!〉〉

return (with ∨ wout)

Con estas elecciones de implementación, el tiempo de ejecución T(n) de nuestro algoritmo satisface la recurrencia T(n) ≤ 2T(n − 1) + O(1). La solución T(n) = O(2ⁿ) se sigue fácilmente usando árboles de recursión o el método aún más simple "Oh sí, ya resolvimos esta recurrencia para la Torre de Hanoi". En el peor caso—por ejemplo, cuando T es mayor que la suma de todos los elementos de X—el árbol de recursión para este algoritmo es un árbol binario completo con profundidad n, y el algoritmo considera todos los 2ⁿ subconjuntos de X.

**Variantes**

Con solo cambios menores, podemos resolver varias variantes de SubsetSum. Por ejemplo, la Figura 2.6 muestra un algoritmo que realmente construye un subconjunto de X que suma T, si existe uno, o devuelve el valor de error None si no existe tal subconjunto; este algoritmo usa exactamente la misma estrategia recursiva que nuestros algoritmos de decisión anteriores. Este algoritmo también ejecuta en tiempo O(2ⁿ); el análisis es más simple si asumimos una estructura de datos de conjunto que nos permite insertar un solo elemento en tiempo O(1) (por ejemplo, una lista enlazada), pero de hecho el tiempo de ejecución sigue siendo O(2ⁿ) incluso si la inserción requiere tiempo O(n) (por ejemplo, una lista enlazada ordenada). Variantes similares nos permiten contar subconjuntos que suman un valor particular, o elegir el mejor subconjunto (según algún otro criterio) que suma un valor particular.

La mayoría de otros problemas que se resuelven mediante retroceso tienen esta propiedad: la misma estrategia recursiva puede usarse para resolver muchas variantes diferentes del mismo problema. Por ejemplo, es fácil modificar la estrategia recursiva descrita en la sección anterior, que determina si una posición de juego dada es buena o mala, para en su lugar devolver un buen movimiento, o una lista de todos los buenos movimientos. Por esta razón, cuando diseñamos algoritmos de retroceso, debemos apuntar a la variante más simple posible del problema, calculando un número o incluso un solo booleano en lugar de información o estructura más compleja.

〈〈Devolver un subconjunto de X[1 .. i] que suma T〉〉

〈〈o NONE si no existe tal subconjunto〉〉

ConstructSubset(X, i, T):

if T = 0

return ∅

if T < 0 or n = 0

return None

Y ← ConstructSubset(X, i − 1, T)

if Y ≠ None

return Y

Y ← ConstructSubset(X, i − 1, T − X[i])

if Y ≠ None

return Y ∪ {X[i]}

return None

**Figura 2.6.** Un algoritmo de retroceso recursivo para la versión de construcción de SUBSETSUM.

**2.4 El Patrón General**

Los algoritmos de retroceso se usan comúnmente para tomar una secuencia de decisiones, con el objetivo de construir una estructura definida recursivamente que satisface ciertas restricciones. A menudo (pero no siempre) esta estructura objetivo es en sí misma una secuencia. Por ejemplo:

• En el problema de las n-reinas, el objetivo es una secuencia de posiciones de reinas, una en cada fila, tal que no hay dos reinas que se ataquen entre sí. Para cada fila, el algoritmo decide dónde colocar la reina.

• En el problema del árbol de juego, el objetivo es una secuencia de movimientos legales, tal que cada movimiento es tan bueno como sea posible para el jugador que lo hace. Para cada estado de juego, el algoritmo decide el mejor movimiento posible siguiente.

• En el problema SubsetSum, el objetivo es una secuencia de elementos de entrada que tienen una suma particular. Para cada elemento de entrada, el algoritmo decide si incluirlo en la secuencia de salida o no.

(Espera, ¿por qué el objetivo de suma de subconjuntos es encontrar una secuencia? Esa fue una decisión de diseño deliberada. Impusimos un ordenamiento conveniente en el conjunto de entrada—representándolo usando un arreglo, en lugar de alguna otra estructura de datos más amorfa—que podemos explotar en nuestro algoritmo recursivo.)

En cada llamada recursiva al algoritmo de retroceso, necesitamos tomar exactamente una decisión, y nuestra elección debe ser consistente con todas las decisiones previas. Así, cada llamada recursiva requiere no solo la porción de los datos de entrada que aún no hemos procesado, sino también un resumen adecuado de las decisiones que ya hemos tomado. Por eficiencia, el resumen de decisiones pasadas debe ser tan pequeño como sea posible. Por ejemplo:

• Para el problema de las n-reinas, debemos pasar no solo el número de filas vacías, sino las posiciones de todas las reinas colocadas previamente. Aquí, desafortunadamente, debemos recordar nuestras decisiones pasadas en detalle completo.

• Para el problema del árbol de juego, solo necesitamos pasar el estado actual del juego, incluyendo la identidad del siguiente jugador. No necesitamos recordar nada sobre nuestras decisiones pasadas, porque quién gana desde un estado de juego dado no depende de los movimientos que crearon ese estado.⁷

• Para el problema SubsetSum, necesitamos pasar tanto los enteros disponibles restantes como el valor objetivo restante, que es el valor objetivo original menos la suma de los elementos elegidos previamente. Precisamente qué elementos fueron elegidos previamente no es importante.

Cuando diseñamos nuevos algoritmos de retroceso recursivos, debemos averiguar de antemano qué información necesitaremos sobre decisiones pasadas en el medio del algoritmo. Si esta información es no trivial, nuestro algoritmo recursivo podría necesitar resolver un problema más general que el que originalmente se nos pidió resolver. (Hemos visto este tipo de generalización antes: Para encontrar la mediana de un arreglo no ordenado en tiempo lineal, derivamos un algoritmo para seleccionar el k-ésimo elemento más pequeño para k arbitrario.)

Finalmente, una vez que hemos averiguado qué problema recursivo realmente necesitamos resolver, resolvemos ese problema mediante fuerza bruta recursiva: Probamos todas las posibilidades para la siguiente decisión que son consistentes con decisiones pasadas, y dejamos que el Hada de la Recursión se preocupe por el resto. No hay que ser inteligente aquí. No hay que omitir elecciones "obviamente" estúpidas. Prueba todo. Puedes hacer el algoritmo más rápido después.

⁷ Muchos juegos parecen violar esta condición de independencia. Por ejemplo, las reglas estándar tanto del ajedrez como de las damas permiten a un jugador declarar empate si la misma disposición de piezas ocurre tres veces, y las reglas chinas para el go simplemente prohíben repetir cualquier disposición anterior de piedras. Así, para estos juegos, un estado de juego incluye formalmente no solo las posiciones actuales de las piezas sino toda la historia de movimientos previos.

**2.5 Segmentación de Texto (Interpunctio Verborum)**

Supón que te dan una cadena de letras que representa texto en algún idioma extranjero, pero sin espacios ni puntuación, y quieres dividir esta cadena en sus palabras constituyentes individuales. Por ejemplo, podrías recibir el siguiente pasaje de la famosa oración de Cicerón en defensa de Lucius Licinius Murena en 62 a.C., en la scriptio continua estándar del latín clásico:⁸

PRIMVSDIGNITASINTAMTENVISCIENTIANONPOTEST ESSERESENIMSVNTPARVAEPROPEINSINGVLISLITTERIS ATQVEINTERPVNCTIONIBVSVERBORVMOCCVPATAE

Un lector fluido de latín analizaría esta cadena (en ortografía moderna) como Primus dignitas in tam tenui scientia non potest esse; res enim sunt parvae, prope in singulis litteris atque interpunctionibus verborum occupatae.⁹ La segmentación de texto no es solo un problema en latín y griego clásicos, sino en varios idiomas y escrituras modernos incluyendo balinés, birmano, chino, japonés, javanés, khmer, lao, tailandés, tibetano y vietnamita. Problemas similares surgen en la segmentación de texto en inglés sin puntuación en oraciones,¹⁰ segmentación de texto en líneas para composición tipográfica, reconocimiento de voz y escritura a mano, simplificación de curvas, y varios tipos de análisis de series temporales. Para propósitos de ilustración, me limitaré a segmentar secuencias de letras en el alfabeto inglés moderno en palabras inglesas modernas.

Por supuesto, algunas cadenas pueden segmentarse de varias maneras diferentes; por ejemplo, BOTHEARTHANDSATURNSPIN puede descomponerse en palabras inglesas como BOTH·EARTH·AND·SATURN·SPIN o BOT·HEART·HANDS·AT·URNS·PIN, entre muchas otras posibilidades. Por ahora, consideremos un problema de segmentación extremadamente simple: Dada una cadena de caracteres, ¿puede segmentarse en palabras inglesas en absoluto?

Para hacer el problema concreto (y agnóstico al idioma), asumamos que tenemos acceso a una subrutina IsWord(w) que toma una cadena w como entrada, y devuelve Verdadero si w es una "palabra", o Falso si w no es una "palabra". Por ejemplo, si estamos tratando de descomponer la cadena de entrada en palíndromos, entonces una "palabra" es sinónimo de "palíndromo", y por lo tanto IsWord(ROTATOR) = Verdadero pero IsWord(PALINDROME) = Falso.

Al igual que el problema SubsetSum, la estructura de entrada es una secuencia, esta vez conteniendo letras en lugar de números, así que es natural considerar un proceso de decisión que consume los caracteres de entrada en orden de izquierda a derecha. Similarmente, la estructura de salida es una secuencia de palabras, así que es natural considerar un proceso que produce las palabras de salida en orden de izquierda a derecha. Así, saltando al medio del proceso de segmentación, podríamos imaginar la siguiente imagen:

BLUE STEM UNIT ROBOT HEARTHANDSATURNSPIN

⁸ De hecho, la mayoría de manuscritos latinos clásicos separaban palabras con pequeños puntos llamados interpuntos. La interpuntuación desapareció casi completamente en el siglo III en favor de la scriptio continua. Los espacios vacíos entre palabras fueron introducidos por monjes irlandeses en el siglo VIII y se extendieron lentamente por Europa durante los siguientes varios siglos. La scriptio continua sobrevive en el inglés de principios del siglo XXI en forma de URLs y hashtags. #octotherps4lyfe

⁹ Traducido libremente: "En primer lugar, la dignidad en tan insignificante conocimiento es imposible; esto es material trivial, principalmente concerniente a letras individuales y la colocación de puntos entre palabras." Cicerón estaba burlándose abiertamente de la expertise legal de su amigo(!) y notable jurista Servius Sulpicius Rufus, quien había acusado a Murena de soborno, después de que Murena derrotara a Rufus en la elección para cónsul. Murena fue absuelto, gracias en parte a la defensa mordaz de Cicerón, aunque casi ciertamente era culpable. #librapondo #nunquamestfidelis

¹⁰ De doctrina Christiana de San Agustín dedica un capítulo entero a remover ambigüedad de las escrituras latinas añadiendo puntuación.

Aquí la barra negra separa nuestras decisiones pasadas—dividir las primeras 17 letras en cuatro palabras—de la porción de la cadena de entrada que aún no hemos procesado. La siguiente etapa en nuestro proceso imaginado es decidir dónde termina la siguiente palabra en la secuencia de salida. Para este ejemplo específico, hay cuatro posibilidades para la siguiente palabra de salida—HE, HEAR, HEART, y HEARTH. No tenemos idea de cuál de estas elecciones, si alguna, es consistente con una segmentación completa de la cadena de entrada. Podríamos ser "inteligentes" en este punto y tratar de averiguar qué elecciones son buenas, pero eso requeriría pensar! En su lugar, "estúpidamente" probemos cada posibilidad por fuerza bruta, y dejemos que el Hada de la Recursión haga todo el trabajo real.

• Primero aceptamos tentativamente HE como la siguiente palabra, y dejamos que el Hada de la Recursión tome el resto de las decisiones.

BLUE STEM UNIT ROBOT HE ARTHANDSATURNSPIN

• Luego aceptamos tentativamente HEAR como la siguiente palabra, y dejamos que el Hada de la Recursión tome el resto de las decisiones.

BLUE STEM UNIT ROBOT HEAR THANDSATURNSPIN

• Luego aceptamos tentativamente HEART como la siguiente palabra, y dejamos que el Hada de la Recursión tome el resto de las decisiones.

BLUE STEM UNIT ROBOT HEART HANDSATURNSPIN

• Finalmente, aceptamos tentativamente HEARTH como la siguiente palabra, y dejamos que el Hada de la Recursión tome el resto de las decisiones.

BLUE STEM UNIT ROBOT HEARTH ANDSATURNSPIN

Mientras el Hada de la Recursión reporte éxito al menos una vez, reportamos éxito. Por otro lado, si el Hada de la Recursión nunca reporta éxito—en particular, si el conjunto de posibles palabras siguientes está vacío—entonces reportamos falla.

Ninguna de nuestras decisiones pasadas afecta qué elecciones están disponibles ahora; todo lo que importa es el sufijo de caracteres que no hemos procesado aún. En particular, varias secuencias diferentes de decisiones pasadas podrían llevarnos al mismo sufijo, pero todas nos dejan con exactamente el mismo conjunto de elecciones para ese sufijo.

BLUE STEM UNIT ROBOT HEARTHANDSATURNSPIN BLUEST EMU NITRO BOT HEARTHANDSATURNSPIN

Así, podemos simplificar nuestra imagen del proceso recursivo descartando todo a la izquierda de la barra negra:

HEARTHANDSATURNSPIN

Ahora nos queda una estrategia de retroceso simple y natural: Seleccionar la primera palabra de salida, y segmentar recursivamente el resto de la cadena de entrada.

Para obtener un algoritmo recursivo completo, necesitamos un caso base. Nuestra estrategia recursiva se descompone cuando llegamos al final de la cadena de entrada, porque no hay siguiente palabra. Afortunadamente, ¡la cadena vacía tiene una segmentación única en cero palabras!

Juntando todas las piezas, llegamos al siguiente algoritmo recursivo simple:

Splittable(A[1 .. n]):

if n = 0

return True

for i ← 1 to n

if IsWord(A[1 .. i])

if Splittable(A[i + 1 .. n])

return True

return False

**Formulación por Índices**

En la práctica, pasar arreglos como parámetros de entrada es bastante lento; realmente deberíamos encontrar una manera más compacta de describir nuestros subproblemas recursivos. Para propósitos de diseñar el algoritmo, es increíblemente útil tratar el arreglo de entrada original como una variable global, y luego reformular el problema y el algoritmo en términos de índices de arreglo en lugar de subarreglos explícitos.

Para nuestro problema de segmentación de cadenas, el argumento de cualquier llamada recursiva es siempre un sufijo A[i .. n] del arreglo de entrada original. Así que si tratamos el arreglo de entrada A[1 .. n] como una variable global, podemos reformular nuestro problema recursivo como sigue:

Dado un índice i, encontrar una segmentación del sufijo A[i .. n].

Para describir nuestro algoritmo, necesitamos dos funciones booleanas:

• Para cualesquiera índices i y j, sea IsWord(i, j) = Verdadero si y solo si la subcadena A[i .. j] es una palabra. (Asumimos que esta función nos es dada.)

• Para cualquier índice i, sea Splittable(i) = Verdadero si y solo si el sufijo A[i .. n] puede dividirse en palabras. (Esta es la función que necesitamos implementar.)

Por ejemplo, IsWord(1, n) = Verdadero si y solo si toda la cadena de entrada es una sola palabra, y Splittable(1) = Verdadero si y solo si toda la cadena de entrada puede segmentarse. Nuestra estrategia recursiva anterior nos da la siguiente recurrencia:

Splittable(i) = { Verdadero si i > n ⋁ⁿⱼ₌ᵢ[IsWord(i, j) ∧ Splittable(j + 1)] de lo contrario }

Este es exactamente el mismo algoritmo que vimos antes; lo único que hemos cambiado es la notación. La similitud es aún más aparente si reescribimos la recurrencia en pseudocódigo:

〈〈¿Es el sufijo A[i .. n] Splittable?〉〉

Splittable(i):

if i > n

return True

for j ← i to n

if IsWord(i, j)

if Splittable(j + 1)

return True

return False

Aunque puede parecer una diferencia notacional trivial, usar notación de índices en lugar de notación de arreglos es un hábito importante, no solo para acelerar algoritmos de retroceso en la práctica, sino para desarrollar algoritmos de programación dinámica, que discutimos en el siguiente capítulo.

**♥Análisis**

No debería sorprender que la mayoría de algoritmos de retroceso tengan tiempos de ejecución exponenciales en el peor caso. Analizar los tiempos de ejecución precisos de muchos de estos algoritmos requiere técnicas que están más allá del alcance de este libro. Afortunadamente, la mayoría de los algoritmos de retroceso que encontraremos en este libro son solo resultados intermedios en el camino hacia algoritmos más eficientes, lo que significa que su tiempo de ejecución exacto en el peor caso en realidad no es importante. (Primero hazlo funcionar; luego hazlo rápido.)

Pero solo por diversión, analicemos el tiempo de ejecución de nuestro algoritmo recursivo Splittable. Porque no sabemos qué está haciendo IsWord, no podemos saber cuánto tiempo toma cada llamada a IsWord, así que nos vemos forzados a analizar el tiempo de ejecución en términos del número de llamadas a IsWord.¹¹ Splittable llama a IsWord en cada prefijo de la cadena de entrada, y posiblemente se llama a sí mismo recursivamente en cada sufijo de la cadena de salida. Así, el "tiempo de ejecución" de Splittable obedece la recurrencia de apariencia aterradora

T(n) ≤ Σⁿ⁻¹ᵢ₌₀ T(i) + O(n)

Esto realmente no es tan malo como parece, especialmente una vez que has visto el truco. Primero, reemplazamos el término O(n) con una expresión explícita αn, para alguna constante desconocida (y finalmente sin importancia) α. Segundo, asumimos conservadoramente que el algoritmo realmente hace cada llamada recursiva posible.¹² Luego podemos transformar la recurrencia de "historia completa" en una recurrencia de "historia limitada" restando la recurrencia para T(n − 1), como sigue:

T(n) = Σⁿ⁻¹ᵢ₌₀ T(i) + αn T(n − 1) = Σⁿ⁻²ᵢ₌₀ T(i) + α(n − 1) ⟹ T(n) − T(n − 1) = T(n − 1) + α

Esta recurrencia final se simplifica a T(n) = 2T(n − 1) + α. En este punto, podemos adivinar confiadamente (o derivar vía árboles de recursión, o recordar de nuestro análisis de Torre de Hanoi) que T(n) = O(2ⁿ); de hecho, esta cota superior no es difícil de probar por inducción desde la recurrencia original de historia completa.

Además, este análisis es ajustado. Hay exactamente 2ⁿ⁻¹ maneras posibles de segmentar una cadena de longitud n—cada carácter de entrada o termina una palabra o no, excepto el último carácter de entrada, que siempre termina la última palabra. En el peor caso, nuestro algoritmo Splittable explora cada una de estas 2ⁿ⁻¹ posibilidades.

¹¹ De hecho, mientras IsWord ejecute en tiempo polinomial, Splittable ejecuta en tiempo O(2ⁿ).

¹² Esta suposición es muy conservadora para segmentación de palabras en inglés, ya que la mayoría de cadenas de letras no son palabras inglesas, pero no para el problema similar de segmentar secuencias de palabras inglesas en oraciones inglesas gramaticalmente correctas. Considera, por ejemplo, una secuencia de n copias de la palabra "buffalo", o n copias de la palabra "police", o n copias de la palabra "can", para cualquier entero positivo n. (En el Moulin Rouge, danzas que son preservables en cilindros metálicos por otras danzas tienen la oportunidad de disparar danzas que ocurren en receptáculos de basura de baños de prisión.)

**Variantes**

Ahora que tenemos el patrón de recursión básico en mano, podemos usarlo para resolver muchas variantes diferentes del problema de segmentación, tal como hicimos para el problema SubsetSum. Aquí describiré solo un ejemplo; más variaciones se consideran en los ejercicios. Como siempre, la entrada original a nuestro problema es un arreglo A[1 .. n].

Si una cadena puede segmentarse en más de una secuencia de palabras, podríamos querer encontrar la mejor segmentación según algún criterio; a la inversa, si la cadena de entrada no puede segmentarse en palabras, podríamos querer calcular la mejor segmentación que podamos encontrar, en lugar de simplemente reportar falla. Para cumplir ambos objetivos, supón que tenemos acceso a una segunda función Score que toma una cadena como entrada y devuelve un valor numérico. Por ejemplo, podríamos asignar puntajes más altos a palabras más largas o más comunes, puntajes más bajos a palabras más cortas u oscuras, puntajes ligeramente negativos para errores ortográficos menores, y puntajes más negativos a obvias no-palabras. Nuestro objetivo es encontrar una segmentación que maximice la suma de los puntajes de los segmentos.

Para cualquier índice i, sea MaxScore(i) el puntaje máximo de cualquier segmentación del sufijo A[i .. n]; necesitamos calcular MaxScore(1). Esta función satisface la siguiente recurrencia:

MaxScore(i) = { 0 si i > n maxᵢ≤ⱼ≤ₙ[Score(A[i .. j]) + MaxScore(j + 1)] de lo contrario }

Esta es esencialmente la misma recurrencia que la que desarrollamos para Splittable; la única diferencia es que las operaciones booleanas ∨ y ∧ han sido reemplazadas por las operaciones numéricas max y +.

**2.6 Subsecuencia Creciente Más Larga**

Para cualquier secuencia S, una subsecuencia de S es otra secuencia obtenida de S eliminando cero o más elementos, sin cambiar el orden de los elementos restantes; los elementos de la subsecuencia no necesitan ser contiguos en S. Por ejemplo, cuando conduces por una calle principal en cualquier ciudad, conduces a través de una secuencia de intersecciones con semáforos, pero solo tienes que parar en una subsecuencia de esas intersecciones, donde los semáforos están en rojo. Si tienes mucha suerte, nunca paras en absoluto: la secuencia vacía es una subsecuencia de S. Por otro lado, si tienes muy mala suerte, puedes tener que parar en cada intersección: S es una subsecuencia de sí misma.

Como otro ejemplo, las cadenas BENT, ACKACK, SQUARING, y SUBSEQUENT son todas subsecuencias de la cadena SUBSEQUENCEBACKTRACKING, como también lo son la cadena vacía y toda la cadena SUBSEQUENCEBACKTRACKING, pero las cadenas QUEUE y EQUUS y TALLYHO no lo son. Una subsecuencia cuyos elementos son contiguos en la secuencia original se llama subcadena; por ejemplo, MASHER y LAUGHTER son ambas subsecuencias de MANSLAUGHTER, pero solo LAUGHTER es una subcadena.

Ahora supón que nos dan una secuencia de enteros, y necesitamos encontrar la subsecuencia más larga cuyos elementos están en orden creciente. Más concretamente, la entrada es un arreglo de enteros A[1 .. n], y necesitamos calcular la secuencia más larga posible de índices 1 ≤ i₁ < i₂ < ··· < i\_ℓ ≤ n tal que A[i\_k] < A[i\_{k+1}] para todo k.

Un enfoque natural para construir esta subsecuencia creciente más larga es decidir, para cada índice j en orden desde 1 hasta n, si incluir o no A[j] en la subsecuencia. Saltando al medio de esta secuencia de decisiones, podríamos imaginar la siguiente imagen:

3 1 4 1 5 9 2 6 5 3 5 ? 8 9 7 9 3 2 3 8 4 6 2 6

Como en nuestros ejemplos de segmentación de texto anteriores, la barra negra separa nuestras decisiones pasadas de la porción de la entrada que aún no hemos procesado. Los números que ya hemos decidido incluir están resaltados y en negrita; los números que ya hemos decidido excluir están en gris. (¡Nota que los números que hemos decidido incluir están creciendo!) Nuestro algoritmo debe decidir si incluir o no el número inmediatamente después de la barra negra.

En este ejemplo, definitivamente no podemos incluir 5, porque entonces los números seleccionados ya no estarían en orden creciente. Así que saltemos al siguiente decisión:

3 1 4 1 5 9 2 6 5 3 5 8 ? 9 7 9 3 2 3 8 4 6 2 6

Ahora podemos incluir 8, pero no es obvio si deberíamos. En lugar de tratar de ser "inteligentes", nuestro algoritmo de retroceso usará fuerza bruta simple.

• Primero incluir tentativamente el 8, y dejar que el Hada de la Recursión tome el resto de las decisiones. • Luego excluir tentativamente el 8, y dejar que el Hada de la Recursión tome el resto de las decisiones.

Cualquier elección que lleve a una subsecuencia creciente más larga es la correcta. (Este es precisamente el mismo patrón de recursión que usamos para resolver SubsetSum.)

Ahora para la pregunta clave: ¿Qué necesitamos recordar sobre nuestras decisiones pasadas? Solo podemos incluir A[j] si la subsecuencia resultante está en orden creciente. Si asumimos (inductivamente!) que los números seleccionados previamente de A[1 .. j − 1] están en orden creciente, entonces podemos incluir A[j] si y solo si A[j] es mayor que el último número seleccionado de A[1 .. j − 1]. Así, la única información que necesitamos sobre el pasado es el último número seleccionado hasta ahora. Ahora podemos revisar nuestras imágenes borrando todo lo que no necesitamos:

6 5 ? 8 9 7 9 3 2 3 8 4 6 2 6 6 8 ? 9 7 9 3 2 3 8 4 6 2 6

Así que el problema que nuestra estrategia recursiva está resolviendo realmente es el siguiente:

Dado un entero prev y un arreglo A[1 .. n], encontrar la subsecuencia creciente más larga de A en la que cada elemento es mayor que prev.

Como siempre, nuestra estrategia recursiva requiere un caso base. Nuestra estrategia actual se descompone cuando llegamos al final del arreglo, porque no hay "siguiente número" que considerar. Pero un arreglo vacío tiene exactamente una subsecuencia, a saber, la secuencia vacía. Vacuamente, cada elemento en la secuencia vacía es mayor que cualquier valor que quieras, y cada par de elementos en la secuencia vacía aparece en orden creciente. Así, la subsecuencia creciente más larga del arreglo vacío tiene longitud 0.

Aquí está el algoritmo recursivo resultante:

LISbigger(prev,A[1 .. n]):

if n = 0

return 0

else if A[1] ≤ prev

return LISbigger(prev,A[2 .. n])

else

skip ← LISbigger(prev,A[2 .. n])

take ← LISbigger(A[1],A[2 .. n]) + 1

return max{skip,take}

Bien, pero recuerda que pasar arreglos en la pila de llamadas es caro; tratemos de reformular todo en términos de índices de arreglo, asumiendo que el arreglo A[1 .. n] es una variable global. El entero prev es típicamente un elemento del arreglo A[i], y el arreglo restante es siempre un sufijo A[j .. n] del arreglo de entrada original. Así que podemos reformular nuestro problema recursivo como sigue:

Dados dos índices i y j, donde i < j, encontrar la subsecuencia creciente más larga de A[j .. n] en la que cada elemento es mayor que A[i].

Sea LISbigger(i, j) la longitud de la subsecuencia creciente más larga de A[j .. n] en la que cada elemento es mayor que A[i]. Nuestra estrategia recursiva nos da la siguiente recurrencia:

LISbigger(i, j) = { 0 si j > n LISbigger(i, j + 1) si A[i] ≥ A[j] max{LISbigger(i, j + 1), 1 + LISbigger(j, j + 1)} de lo contrario }

Alternativamente, si prefieres pseudocódigo:

LISbigger(i, j):

if j > n

return 0

else if A[i] ≥ A[j]

return LISbigger(i, j + 1)

else

skip ← LISbigger(i, j + 1)

take ← LISbigger(j, j + 1) + 1

return max{skip,take}

Finalmente, necesitamos conectar nuestra estrategia recursiva al problema original: Encontrar la subsecuencia creciente más larga de un arreglo sin otras restricciones. El enfoque más simple es añadir un valor centinela artificial −∞ al comienzo del arreglo.

LIS(A[1 .. n]):

A[0] ← −∞

return LISbigger(0, 1)

El tiempo de ejecución de LISbigger satisface la recurrencia de Hanoi T(n) ≤ 2T(n−1)+O(1), que como siempre implica que T(n) = O(2ⁿ). Realmente no deberíamos sorprendernos por este tiempo de ejecución; en el peor caso, el algoritmo examina cada una de las 2ⁿ subsecuencias del arreglo de entrada.

**2.7 Subsecuencia Creciente Más Larga, Toma 2**

Esta no es la única estrategia de retroceso que podemos usar para encontrar subsecuencias crecientes más largas. En lugar de considerar la secuencia de entrada un elemento a la vez, podríamos tratar de construir la secuencia de salida un elemento a la vez. Es decir, en lugar de preguntar "¿Es A[i] el siguiente elemento de la secuencia de salida?", podríamos preguntar directamente, "¿Dónde está el siguiente elemento de la secuencia de salida, si lo hay?"

Saltando al medio de esta estrategia, podríamos estar enfrentados con la siguiente imagen. Supón que acabamos de decidir incluir el 6 justo a la izquierda de la barra negra en nuestra secuencia de salida, y necesitamos decidir qué elemento a la derecha de la barra incluir después.

3 1 4 1 5 9 2 6 | 5? 3? 5? 8? 9? 7? 9? 3? 2? 3? 8? 4? 6? 2? 6?

Por supuesto, solo podemos incluir números a la derecha que sean mayores que 6; de lo contrario, nuestra secuencia de salida no sería creciente.

3 1 4 1 5 9 2 6 | 5 3 5 8? 9? 7? 9? 3 2 3 8? 4 6 2 6

Pero no tenemos idea de cuál de esos números mayores es la mejor elección, y tratar de averiguar inteligentemente la mejor elección es demasiado trabajo, y solo nos va a meter en problemas de todos modos. En su lugar, enumeramos todas las posibilidades por fuerza bruta, y dejamos que el Hada de la Recursión evalúe cada una.

3 1 4 1 5 9 2 6 | 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 3 1 4 1 5 9 2 6 | 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 3 1 4 1 5 9 2 6 | 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 3 1 4 1 5 9 2 6 | 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 3 1 4 1 5 9 2 6 | 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6

El subconjunto de números que podemos considerar como el siguiente elemento depende solo del último número que decidimos incluir. Así, podemos simplificar nuestra imagen del proceso de decisión descartando todo a la izquierda de la barra excepto el último número que decidimos incluir.

6 | 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6

La secuencia restante de números es solo un sufijo del arreglo de entrada original. Así, si pensamos en el arreglo de entrada A[1 .. n] como una variable global, podemos expresar formalmente nuestro problema recursivo en términos de índices como sigue:

Dado un índice i, encontrar la subsecuencia creciente más larga de A[i .. n] que comienza con A[i].

Sea LISfirst(i) la longitud de la subsecuencia creciente más larga de A[i .. n] que comienza con A[i]. Ahora podemos formular nuestra estrategia de retroceso recursiva como la siguiente definición recursiva:

LISfirst(i) = 1 + max{LISfirst(j) | j > i y A[j] > A[i]}

Porque estamos tratando con conjuntos de números naturales, definimos max∅ = 0. Luego automáticamente tenemos LISfirst(i) = 1 si A[j] ≤ A[i] para todo j > i; en particular, LISfirst(n) = 1. Estos son los casos base para nuestra recurrencia.

También podemos expresar esta definición recursiva en pseudocódigo como sigue:

LISfirst(i):

best ← 0

for j ← i + 1 to n

if A[j] > A[i]

best ← max{best, LISfirst(j)}

return 1 + best

Finalmente, necesitamos reconectar este algoritmo recursivo a nuestro problema original—encontrar la subsecuencia creciente más larga sin conocer su primer elemento. Un enfoque natural que funciona es probar todos los primeros elementos posibles por fuerza bruta. Equivalentemente, podemos añadir un elemento centinela −∞ al comienzo del arreglo, encontrar la subsecuencia creciente más larga que comienza con el centinela, y finalmente ignorar el centinela.

LIS(A[1 .. n]):

best ← 0

for i ← 1 to n

best ← max{best, LISfirst(i)}

return best

LIS(A[1 .. n]):

A[0] ← −∞

return LISfirst(0) − 1

**2.8 Árboles de Búsqueda Binaria Óptimos**

Nuestro ejemplo final combina retroceso recursivo con la estrategia de divide y vencerás. Recuerda que el tiempo de ejecución para una búsqueda exitosa en un árbol de búsqueda binaria es proporcional al número de ancestros del nodo objetivo.¹³ Como resultado, el tiempo de búsqueda en el peor caso es proporcional a la profundidad del árbol. Así, para minimizar el tiempo de búsqueda en el peor caso, la altura del árbol debe ser tan pequeña como sea posible; por esta métrica, el árbol ideal está perfectamente balanceado.

En muchas aplicaciones de árboles de búsqueda binaria, sin embargo, es más importante minimizar el costo total de varias búsquedas en lugar del costo en el peor caso de una sola búsqueda. Si x es un objetivo de búsqueda más frecuente que y, podemos ahorrar tiempo construyendo un árbol donde la profundidad de x es menor que la profundidad de y, incluso si eso significa aumentar la profundidad general del árbol. Un árbol perfectamente balanceado no es la mejor elección si algunos elementos son significativamente más populares que otros. De hecho, ¡un árbol totalmente desbalanceado con profundidad Ω(n) podría ser realmente la mejor elección!

Esta situación sugiere el siguiente problema. Supón que nos dan un arreglo ordenado de claves A[1 .. n] y un arreglo de frecuencias de acceso correspondientes f[1 .. n]. Nuestra tarea es construir el árbol de búsqueda binaria que minimiza el tiempo total de búsqueda, asumiendo que habrá exactamente f[i] búsquedas para cada clave A[i].

Antes de pensar en cómo resolver este problema, ¡primero deberíamos llegar a una buena definición recursiva de la función que estamos tratando de optimizar! Supón que también nos dan un árbol de búsqueda binaria T con n nodos. Sean v₁, v₂, . . . , vₙ los nodos de T, indexados en orden ordenado, de manera que cada nodo vᵢ almacena la clave correspondiente A[i]. Luego ignorando factores constantes, el costo total de realizar todas las búsquedas binarias está dado por la siguiente expresión:

Cost(T, f[1 .. n]) := Σᵢ₌₁ⁿ f[i] · #ancestros de vᵢ en T (\*)

Ahora supón que vᵣ es la raíz de T; por definición, vᵣ es un ancestro de cada nodo en T. Si i < r, entonces todos los ancestros de vᵢ excepto la raíz están en el subárbol izquierdo de T. Similarmente, si i > r, entonces todos los ancestros de vᵢ excepto la raíz están en el subárbol derecho de T. Así, podemos particionar la función de costo en tres partes como sigue:

Cost(T, f[1 .. n]) = Σᵢ₌₁ⁿ f[i] + Σᵢ₌₁ʳ⁻¹ f[i] · #ancestros de vᵢ en left(T) + Σᵢ₌ᵣ₊₁ⁿ f[i] · #ancestros de vᵢ en right(T)

Las segunda y tercera sumatorias se ven exactamente como nuestra definición original (\*) para Cost(T, f[1 .. n]). La substitución simple ahora nos da una recurrencia para Cost:

Cost(T, f[1 .. n]) = Σᵢ₌₁ⁿ f[i] + Cost(left(T), f[1 ..r − 1]) + Cost(right(T), f[r + 1 .. n])

El caso base para esta recurrencia es, como siempre, n = 0; el costo de realizar cero búsquedas en el árbol vacío es cero.

Ahora nuestra tarea es calcular el árbol Tₒₚₜ que minimiza esta función de costo. Supón que de alguna manera sabemos mágicamente que la raíz de Tₒₚₜ es vᵣ. Entonces la definición recursiva de Cost(T, f) implica inmediatamente que el subárbol izquierdo left(Tₒₚₜ) debe ser el árbol de búsqueda óptimo para las claves A[1 ..r − 1] y frecuencias de acceso f[1 ..r − 1]. Similarmente, el subárbol derecho right(Tₒₚₜ) debe ser el árbol de búsqueda óptimo para las claves A[r + 1 .. n] y frecuencias de acceso f[r + 1 .. n]. Una vez que elegimos la clave correcta para almacenar en la raíz, el Hada de la Recursión construirá el resto del árbol óptimo.

Más generalmente, sea OptCost(i, k) el costo total del árbol de búsqueda óptimo para el intervalo de frecuencias f[i .. k]. Esta función obedece la siguiente recurrencia:

OptCost(i, k) = { 0 si i > k Σⱼ₌ᵢᵏ f[i] + min\_{i≤r≤k}[OptCost(i,r − 1) + OptCost(r + 1, k)] de lo contrario }

¡El caso base indica correctamente que el costo mínimo posible para realizar cero búsquedas en el conjunto vacío es cero! Nuestro problema original es calcular OptCost(1, n).

Esta definición recursiva puede traducirse mecánicamente en un algoritmo de retroceso recursivo para calcular OptCost(1, n). No es sorprendente que el tiempo de ejecución de este algoritmo sea exponencial. En el siguiente capítulo, veremos cómo reducir el tiempo de ejecución a polinomial, así que no hay mucho punto en calcular el tiempo de ejecución preciso...

¹³ Un ancestro de un nodo v es o el nodo mismo o un ancestro del padre de v. Un ancestro propio de v es o el padre de v o un ancestro propio del padre de v.

**♥Análisis**

... a menos que te guste ese tipo de cosas. Solo por diversión, averigüemos qué tan lento es realmente este algoritmo de retroceso. El tiempo de ejecución satisface la recurrencia

T(n) = Σₖ₌₁ⁿ [T(k − 1) + T(n − k)] + O(n).

El término O(n) viene de calcular el número total de búsquedas Σᵢ₌₁ⁿ f[i]. Sí, esa es una recurrencia fea, pero podemos resolverla usando exactamente el mismo truco de sustracción que usamos antes. Reemplazamos la notación O( ) con una constante explícita, reagrupamos y colectamos términos idénticos, restamos la recurrencia para T(n−1) para deshacernos de la sumatoria, y luego reagrupamos otra vez.

T(n) = 2Σₖ₌₀ⁿ⁻¹ T(k) + αn T(n − 1) = 2Σₖ₌₀ⁿ⁻² T(k) + α(n − 1) T(n) − T(n − 1) = 2T(n − 1) + α T(n) = 3T(n − 1) + α

¡Oye, eso no se ve tan mal después de todo! El método del árbol de recursión inmediatamente nos da la solución T(n) = O(3ⁿ) (o podemos simplemente adivinar y confirmar por inducción).

Este análisis implica que nuestro algoritmo recursivo no examina todos los árboles de búsqueda binaria posibles! El número de árboles de búsqueda binaria con n vértices satisface la recurrencia

N(n) = Σᵣ₌₁ⁿ⁻¹ [N(r − 1)· N(n − r)]

que tiene la solución de forma cerrada N(n) = Θ(4ⁿ/√n). (No, eso no es obvio.) Nuestro algoritmo ahorra tiempo considerable buscando independientemente los subárboles izquierdo y derecho óptimos para cada raíz. Una enumeración completa de árboles de búsqueda binaria consideraría todos los pares posibles de subárboles izquierdo y derecho; de ahí el producto en la recurrencia para N(n).

**Ejercicios**

1. Describe algoritmos recursivos para las siguientes generalizaciones del problema SubsetSum:

(a) Dados un arreglo X[1 .. n] de enteros positivos y un entero T, calcular el número de subconjuntos de X cuyos elementos suman T.

(b) Dados dos arreglos X[1 .. n] y W[1 .. n] de enteros positivos y un entero T, donde cada W[i] denota el peso del elemento correspondiente X[i], calcular el subconjunto de peso máximo de X cuyos elementos suman T. Si ningún subconjunto de X suma T, tu algoritmo debe devolver −∞.

1. Describe algoritmos recursivos para las siguientes variantes del problema de segmentación de texto. Asume que tienes una subrutina IsWord que toma un arreglo de caracteres como entrada y devuelve Verdadero si y solo si esa cadena es una "palabra".

(a) Dado un arreglo A[1 .. n] de caracteres, calcular el número de particiones de A en palabras. Por ejemplo, dada la cadena ARTISTOIL, tu algoritmo debe devolver 2, para las particiones ARTIST·OIL y ART·IS·TOIL.

(b) Dados dos arreglos A[1 .. n] y B[1 .. n] de caracteres, decidir si A y B pueden particionarse en palabras en los mismos índices. Por ejemplo, las cadenas BOTHEARTHANDSATURNSPIN y PINSTARTRAPSANDRAGSLAP pueden particionarse en palabras en los mismos índices como sigue: BOT·HEART·HAND·SAT·URNS·PIN PIN·START·RAPS·AND·RAGS·LAP

(c) Dados dos arreglos A[1 .. n] y B[1 .. n] de caracteres, calcular el número de maneras diferentes que A y B pueden particionarse en palabras en los mismos índices.

1. Una cadena de adición para un entero n es una secuencia creciente de enteros que comienza con 1 y termina con n, tal que cada entrada después de la primera es la suma de dos entradas anteriores. Más formalmente, la secuencia de enteros x₀ < x₁ < x₂ < ··· < x\_ℓ es una cadena de adición para n si y solo si

• x₀ = 1, • x\_ℓ = n, y • para cada índice k > 0, hay índices i ≤ j < k tales que x\_k = x\_i + x\_j.

La longitud de una cadena de adición es el número de elementos menos 1; no nos molestamos en contar la primera entrada. Por ejemplo, ⟨1, 2, 3, 5, 10, 20, 23, 46, 92, 184, 187, 374⟩ es una cadena de adición para 374 de longitud 11.

(a) Describe un algoritmo de retroceso recursivo para calcular una cadena de adición de longitud mínima para un entero positivo dado n. No analices u optimices el tiempo de ejecución de tu algoritmo, excepto para satisfacer tu propia curiosidad. Un algoritmo correcto cuyo tiempo de ejecución sea exponencial en n es suficiente para crédito completo. [Pista: Este problema es mucho más como n Reinas que segmentación de texto.]

♥(b) Describe un algoritmo de retroceso recursivo para calcular una cadena de adición de longitud mínima para un entero positivo dado n en tiempo que sea sub-exponencial en n. [Pista: Puedes encontrar útiles los resultados de ciertos atadores de cuerdas egipcios, prosodistas del Río Indo, y campesinos rusos.]

1. (a) Sean A[1 .. m] y B[1 .. n] dos arreglos arbitrarios. Una subsecuencia común de A y B es tanto una subsecuencia de A como una subsecuencia de B. Da una definición recursiva simple para la función lcs(A, B), que da la longitud de la subsecuencia común más larga de A y B.

(b) Sean A[1 .. m] y B[1 .. n] dos arreglos arbitrarios. Una supersecuencia común de A y B es otra secuencia que contiene tanto A como B como subsecuencias. Da una definición recursiva simple para la función scs(A, B), que da la longitud de la supersecuencia común más corta de A y B.

(c) Llama a una secuencia X[1 .. n] de números bitónica si hay un índice i con 1 < i < n, tal que el prefijo X[1 .. i] es creciente y el sufijo X[i .. n] es decreciente. Da una definición recursiva simple para la función lbs(A), que da la longitud de la subsecuencia bitónica más larga de un arreglo arbitrario A de enteros.

(d) Llama a una secuencia X[1 .. n] oscilante si X[i] < X[i + 1] para todo i par, y X[i] > X[i + 1] para todo i impar. Da una definición recursiva simple para la función los(A), que da la longitud de la subsecuencia oscilante más larga de un arreglo arbitrario A de enteros.

(e) Da una definición recursiva simple para la función sos(A), que da la longitud de la supersecuencia oscilante más corta de un arreglo arbitrario A de enteros.

(f) Llama a una secuencia X[1 .. n] convexa si 2 · X[i] < X[i − 1] + X[i + 1] para todo i. Da una definición recursiva simple para la función lxs(A), que da la longitud de la subsecuencia convexa más larga de un arreglo arbitrario A de enteros.

1. Para cada uno de los siguientes problemas, la entrada consiste en dos arreglos X[1 .. k] y Y[1 .. n] donde k ≤ n.

(a) Describe un algoritmo de retroceso recursivo para determinar si X es una subsecuencia de Y. Por ejemplo, la cadena PPAP es una subsecuencia de la cadena PENPINEAPPLEAPPLEPEN.

(b) Describe un algoritmo de retroceso recursivo para encontrar el menor número de símbolos que pueden removerse de Y de manera que X ya no sea una subsecuencia. Equivalentemente, tu algoritmo debe encontrar la subsecuencia más larga de Y que no sea una supersecuencia de X. Por ejemplo, después de remover dos símbolos de la cadena PENPINEAPPLEAPPLEPEN, la cadena PPAP ya no es una subsecuencia.

♥(c) Describe un algoritmo de retroceso recursivo para determinar si X ocurre como dos subsecuencias disjuntas de Y. Por ejemplo, la cadena PPAP aparece como dos subsecuencias disjuntas en la cadena PENPINEAPPLEAPPLEPEN.

No analices los tiempos de ejecución de tus algoritmos, excepto para satisfacer tu propia curiosidad. Los tres algoritmos ejecutan en tiempo exponencial; mejoraremos eso más tarde, así que el tiempo de ejecución preciso no es particularmente importante.

1. Este problema te pide diseñar algoritmos de retroceso para encontrar el costo de un árbol de búsqueda binaria óptimo que satisface restricciones de balance adicionales. Tu entrada consiste en un arreglo ordenado A[1 .. n] de claves de búsqueda y un arreglo f[1 .. n] de conteos de frecuencia, donde f[i] es el número de búsquedas para A[i]. Esta es exactamente la misma función de costo descrita en la Sección 2.8. Pero ahora tu tarea es calcular un árbol óptimo que satisface algunas restricciones adicionales.

(a) Los árboles AVL fueron los primeros árboles de búsqueda binaria auto-balanceados, descritos por primera vez en 1962 por Georgy Adelson-Velsky y Evgenii Landis. Un árbol AVL es un árbol de búsqueda binaria donde para cada nodo v, la altura del subárbol izquierdo de v y la altura del subárbol derecho de v difieren en a lo más uno.

Describe un algoritmo de retroceso recursivo para construir un árbol AVL óptimo para un conjunto dado de claves de búsqueda y frecuencias.

(b) Los árboles B binarios simétricos son otros árboles binarios auto-balanceados, descritos por primera vez por Rudolf Bayer en 1972; estos son mejor conocidos por el nombre árboles rojo-negro, después de una reformulación algo más simple por Leo Guibas y Bob Sedgwick en 1978. Un árbol rojo-negro es un árbol de búsqueda binaria con las siguientes restricciones adicionales:

• Cada nodo es rojo o negro.

• Cada nodo rojo tiene un padre negro.

• Cada camino de raíz a hoja contiene el mismo número de nodos negros.

Describe un algoritmo de retroceso recursivo para construir un árbol rojo-negro óptimo para un conjunto dado de claves de búsqueda y frecuencias.

(c) Los árboles AA fueron propuestos por Arne Andersson en 1993 y ligeramente simplificados (y nombrados) por Mark Allen Weiss en 2000. Los árboles AA también son conocidos como árboles rojo-negro inclinados a la izquierda, después de una reformulación simétrica (con diferentes algoritmos de rebalanceo) por Bob Sedgwick en 2006. Un árbol AA es un árbol rojo-negro con una restricción adicional:

• Ningún hijo izquierdo es rojo.¹⁴

Describe un algoritmo de retroceso recursivo para construir un árbol AA óptimo para un conjunto dado de claves de búsqueda y frecuencias.

No analices los tiempos de ejecución de tus algoritmos, excepto para satisfacer tu propia curiosidad. Los tres algoritmos ejecutan en tiempo exponencial; mejoraremos eso más tarde, así que los tiempos de ejecución precisos no son particularmente importantes.

¹⁴ La reformulación de Sedgwick requiere que ningún hijo derecho sea rojo. Lo que sea. Andersson y Sedgwick están extrañamente silenciosos sobre qué extremo del huevo comer primero.

**Para más ejercicios de retroceso, ¡ve el siguiente capítulo!**

**Capítulo 3: Programación Dinámica**

*Puedes ver en este margen cómo hemos trabajado esto; claramente, combinamos el primer número con el segundo, es decir, 1 con 2, y el segundo con el tercero, y el tercero con el cuarto, y el cuarto con el quinto, y así sucesivamente...* — Leonardo Pisano, Liber Abaci (1202)

*Aquellos que no pueden recordar el pasado están condenados a repetirlo.* — Jorge Agustín Nicolás Ruiz de Santayana y Borrás, La Vida de la Razón, Libro I: Introducción y Razón en el Sentido Común (1905)

*¿Sabes qué es una experiencia de aprendizaje? Una experiencia de aprendizaje es una de esas cosas que dice: "¿Sabes esa cosa que acabas de hacer? No hagas eso."* — Douglas Adams, El Salmón de la Duda (2002)

**3.1 Mātravṛtta**

Uno de los primeros ejemplos de recursión surgió en la India hace más de 2000 años, en el estudio de la métrica poética, o prosodia. La poesía sánscrita clásica distingue entre dos tipos de sílabas (akṣara): ligeras (laghu) y pesadas (guru). En una clase de metros, llamados diversamente mātravṛtta o mātrāchandas, cada línea de poesía consiste en un número fijo de "tiempos" (mātrā), donde cada sílaba ligera dura un tiempo y cada sílaba pesada dura dos tiempos. El estudio formal del mātravṛtta se remonta al Chandaḥśāstra, escrito por el erudito Piṅgala entre 600 a.C. y 200 a.C. Piṅgala observó que hay exactamente cinco metros de 4 tiempos: ——, —••, •—•, ••—, y ••••. (Aquí cada "—" representa una sílaba larga y cada "•" representa una sílaba corta.)¹

Aunque el texto de Piṅgala sugiere una regla sistemática para contar metros con un número dado de tiempos,² tomó aproximadamente un milenio para que esa regla fuera declarada explícitamente. En el siglo VII d.C., otro erudito indio llamado Virahaṅka escribió un comentario sobre el trabajo de Piṅgala, en el cual observó que el número de metros con n tiempos es la suma del número de metros con (n − 2) tiempos y el número de metros con (n − 1) tiempos. En notación más moderna, la observación de Virahaṅka implica una recurrencia para el número total M(n) de metros de n tiempos:

M(n) = M(n − 2) + M(n − 1)

No es difícil ver que M(0) = 1 (hay solo un metro vacío) y M(1) = 1 (el único metro de un tiempo consiste en una sola sílaba corta).

La misma recurrencia reapareció en Europa aproximadamente 500 años después de Virahaṅka, en el tratado de 1202 de Leonardo de Pisa Liber Abaci, una de las obras europeas tempranas más influyentes sobre "algorismo". En pleno cumplimiento con la Ley de Eponimia de Stigler,³ los números de Fibonacci modernos se definen usando la recurrencia de Virahaṅka, pero con casos base diferentes:

Fₙ = { 0 si n = 0 1 si n = 1 Fₙ₋₁ + Fₙ₋₂ en caso contrario }

En particular, tenemos M(n) = Fₙ₊₁ para todo n.

**El Retroceso Puede Ser Lento**

La definición recursiva de los números de Fibonacci inmediatamente nos da un algoritmo recursivo para calcularlos. Aquí está el mismo algoritmo escrito en pseudocódigo:

RecFibo(n):

si n = 0

retornar 0

sino si n = 1

retornar 1

sino

retornar RecFibo(n − 1) + RecFibo(n − 2)

Desafortunadamente, este algoritmo recursivo ingenuo es horriblemente lento. Excepto por las llamadas recursivas, todo el algoritmo requiere solo un número constante de pasos: una comparación y posiblemente una suma. Sea T(n) el número de llamadas recursivas a RecFibo; esta función satisface la recurrencia

T(0) = 1, T(1) = 1, T(n) = T(n − 1) + T(n − 2) + 1,

¡que se parece mucho a la recurrencia para los números de Fibonacci mismos! Escribir los primeros varios valores de T(n) sugiere la solución de forma cerrada T(n) = 2Fₙ₊₁ − 1, que podemos verificar por inducción (pista, pista). Así que calcular Fₙ usando este algoritmo toma aproximadamente el doble del tiempo que simplemente contar hasta Fₙ. Métodos más allá del alcance de este libro⁴ implican que Fₙ = Θ(φⁿ), donde φ = (√5 + 1)/2 ≈ 1.61803 es la llamada razón dorada. En resumen, el tiempo de ejecución de este algoritmo recursivo es exponencial en n.

**Referencia a Figura 3.1**: El árbol de recursión para calcular F₇; las flechas representan llamadas recursivas.

Podemos ver este crecimiento exponencial directamente de la siguiente manera. Piensa en el árbol de recursión para RecFibo como un árbol binario de sumas, con solo 0s y 1s en las hojas. Como la salida eventual es Fₙ, exactamente Fₙ de las hojas deben tener valor 1; estas hojas representan las llamadas a RecFibo(1). Un argumento inductivo fácil (pista, pista) implica que RecFibo(0) se llama exactamente Fₙ₋₁ veces. (Si solo queremos una cota asintótica, es suficiente observar que el número de llamadas a RecFibo(0) es a lo sumo el número de llamadas a RecFibo(1).) Así, el árbol de recursión tiene exactamente Fₙ + Fₙ₋₁ = Fₙ₊₁ = O(Fₙ) hojas, y por lo tanto, porque es un árbol binario completo, 2Fₙ₊₁ − 1 = O(Fₙ) nodos en total.

**Memo(r)ización: Recordar Todo**

La razón obvia para la falta de velocidad del algoritmo recursivo es que calcula los mismos números de Fibonacci una y otra y otra vez. Una sola llamada a RecFibo(n) resulta en una llamada recursiva a RecFibo(n − 1), dos llamadas recursivas a RecFibo(n − 2), tres llamadas recursivas a RecFibo(n − 3), cinco llamadas recursivas a RecFibo(n − 4), y en general Fₖ₋₁ llamadas recursivas a RecFibo(n − k) para cualquier entero 0 ≤ k < n. Cada llamada está recalculando algún número de Fibonacci desde cero.

Podemos acelerar nuestro algoritmo recursivo considerablemente escribiendo los resultados de nuestras llamadas recursivas y buscándolos de nuevo si los necesitamos más tarde.

MemFibo(n):

si n = 0

retornar 0

sino si n = 1

retornar 1

sino

si F[n] está indefinido

F[n] ← MemFibo(n − 1) + MemFibo(n − 2)

retornar F[n]

Esta técnica de optimización, ahora conocida como memoización (sí, sin una R), usualmente se atribuye a Donald Michie en 1967, pero esencialmente la misma técnica fue propuesta en 1959 por Arthur Samuel.⁵

**Referencia a Figura 3.2**: El árbol de recursión para F₇ recortado por memoización. Las flechas verdes hacia abajo indican escritura en el arreglo de memoización; las flechas rojas hacia arriba indican lectura del arreglo de memoización.

La memoización claramente disminuye el tiempo de ejecución del algoritmo, ¿pero por cuánto? Si realmente rastreamos las llamadas recursivas hechas por MemFibo, encontramos que el arreglo F[ ] se llena de abajo hacia arriba: primero F[2], luego F[3], y así sucesivamente, hasta F[n]. Este patrón puede verificarse por inducción: Cada entrada F[i] se llena solo después de su predecesor F[i − 1]. Si ignoramos el tiempo gastado en llamadas recursivas, requiere solo tiempo constante evaluar la recurrencia para cada número de Fibonacci Fᵢ. Pero por diseño, la recurrencia para Fᵢ se evalúa solo una vez para cada índice i. Concluimos que MemFibo realiza solo O(n) sumas, ¡una mejora exponencial sobre el algoritmo recursivo ingenuo!

**Programación Dinámica: Llenar Deliberadamente**

Una vez que vemos cómo se llena el arreglo F[ ], podemos reemplazar la recurrencia memoizada con un bucle for simple que intencionalmente llene el arreglo en ese orden, en lugar de depender de un algoritmo recursivo más complicado para hacerlo por nosotros accidentalmente.

IterFibo(n):

F[0] ← 0

F[1] ← 1

para i ← 2 hasta n

F[i] ← F[i − 1] + F[i − 2]

retornar F[n]

Ahora el análisis de tiempo es inmediato: IterFibo claramente usa O(n) sumas y almacena O(n) enteros.

Este es nuestro primer algoritmo explícito de programación dinámica. El paradigma de programación dinámica fue formalizado y popularizado por Richard Bellman a mediados de los años 1950, mientras trabajaba en la Corporación RAND, aunque estaba lejos de ser el primero en usar la técnica. En particular, este algoritmo iterativo para números de Fibonacci ya fue propuesto por Virahaṅka y prosodistas sánscritos posteriores en el siglo XII, ¡y de nuevo por Fibonacci al final del siglo XIII!⁶

Muchos años después del hecho, Bellman afirmó que deliberadamente eligió el nombre "programación dinámica" para ocultar el carácter matemático de su trabajo de sus jefes militares, que eran activamente hostiles hacia cualquier cosa que se pareciera a investigación matemática.⁷ La palabra "programación" no se refiere a escribir código, sino más bien al sentido más antiguo de planificación o programación, típicamente llenando una tabla. Por ejemplo, los programas deportivos y teatrales son horarios de eventos importantes (con anuncios); la programación televisiva involucra llenar cada espacio de tiempo disponible con un show (y anuncios); los programas de grado son horarios de clases a tomar (con anuncios). La Fuerza Aérea financió a Bellman y otros para desarrollar métodos para construir horarios de entrenamiento y logística, o como los llamaban, "programas". La palabra "dinámico" no solo era una referencia a los procesos multietapa que varían en el tiempo que Bellman y sus colegas estaban intentando optimizar, sino también una palabra de moda de marketing que resonaría con el Zeitgeist™ Futurista Can-Do de la América posterior a la Segunda Guerra Mundial.⁸ Gracias en parte al proselitismo de Bellman, la programación dinámica es ahora una herramienta estándar para planificación multietapa en economía, robótica, teoría de control y varias otras disciplinas.

**No Recordar Todo Después de Todo**

En muchos algoritmos de programación dinámica, no es necesario retener todos los resultados intermedios durante toda la computación. Por ejemplo, podemos reducir significativamente los requisitos de espacio de nuestro algoritmo IterFibo manteniendo solo los dos elementos más nuevos del arreglo:

IterFibo2(n):

prev ← 1

curr ← 0

para i ← 1 hasta n

next ← curr + prev

prev ← curr

curr ← next

retornar curr

(Este algoritmo usa el caso base no estándar pero consistente F₋₁ = 1 para que IterFibo2(0) retorne el valor correcto 0.) Aunque ahorrar espacio puede ser absolutamente crucial en la práctica, no nos enfocaremos en problemas de espacio en este libro.

**♥3.2 Aparte: Números de Fibonacci Aún Más Rápidos**

Aunque el algoritmo anterior es simple y atractivo, no es el algoritmo más rápido para calcular números de Fibonacci. Podemos derivar un algoritmo más rápido explotando la siguiente reformulación matricial de la recurrencia de Fibonacci:

[0 1] [x] = [y ] [1 1] [y] [x + y]

En otras palabras, multiplicar un vector bidimensional por la matriz [0 1; 1 1] tiene exactamente el mismo efecto que una iteración del bucle interno de IterFibo2. Se sigue que multiplicar por la matriz n veces es lo mismo que iterar el bucle n veces:

[0 1]ⁿ [1] = [Fₙ₋₁] [1 1] [0] [Fₙ ]

Así que si queremos el n-ésimo número de Fibonacci, solo necesitamos calcular la n-ésima potencia de la matriz [0 1; 1 1]. Si usamos elevación al cuadrado repetida,⁹ calcular la n-ésima potencia de algo requiere solo O(log n) multiplicaciones. Aquí, porque "algo" es una matriz de 2 × 2, eso significa O(log n) multiplicaciones de matrices de 2 × 2, cada una de las cuales se reduce a un número constante de multiplicaciones y sumas de enteros. Así, podemos calcular Fₙ en solo O(log n) operaciones aritméticas de enteros.

Podemos lograr la misma aceleración usando la identidad Fₙ = FₘFₙ₋ₘ₋₁ + Fₘ₊₁Fₙ₋ₘ, que se cumple (¡por inducción!) para todos los enteros m y n. En particular, esta identidad implica la siguiente recurrencia mutua para pares de números de Fibonacci adyacentes, propuesta por primera vez por Édouard Lucas en 1898:

F₂ₙ₋₁ = F²ₙ₋₁ + F²ₙ F₂ₙ = Fₙ(Fₙ₋₁ + Fₙ₊₁) = Fₙ(2Fₙ₋₁ + Fₙ)

(También podemos derivar esta recurrencia mutua directamente del algoritmo de elevación al cuadrado de matrices.) Estas recurrencias se traducen directamente en el siguiente algoritmo:

⟨⟨Calcular el par Fₙ₋₁, Fₙ⟩⟩

FastRecFibo(n):

si n = 1

retornar 0, 1

m ← ⌊n/2⌋

hprv, hcur ← FastRecFibo(m) ⟨⟨Fₘ₋₁, Fₘ⟩⟩

prev ← hprv² + hcur² ⟨⟨F₂ₘ₋₁⟩⟩

curr ← hcur·(2 · hprv + hcur) ⟨⟨F₂ₘ⟩⟩

next ← prev + curr ⟨⟨F₂ₘ₊₁⟩⟩

si n es par

retornar prev, curr

sino

retornar curr, next

Nuestra técnica estándar de árbol de recursión implica que este algoritmo realiza solo O(log n) operaciones aritméticas de enteros.

Esta es una aceleración exponencial sobre el algoritmo iterativo estándar, que ya era una aceleración exponencial sobre nuestro algoritmo recursivo original. ¿Correcto?

¡Whoa! ¡No tan rápido!

Bueno, no exactamente. Los números de Fibonacci crecen exponencialmente rápido. El n-ésimo número de Fibonacci es aproximadamente n log₁₀ φ ≈ n/5 dígitos decimales de largo, o n log₂ φ ≈ 2n/3 bits. ¡Así que no podemos posiblemente calcular Fₙ en tiempo logarítmico — necesitamos tiempo Ω(n) solo para escribir la respuesta!

La salida de esta aparente paradoja es observar que no podemos realizar aritmética de precisión arbitraria en tiempo constante. Sea M(n) el tiempo requerido para multiplicar dos números de n dígitos. El tiempo de ejecución de FastRecFibo satisface la recurrencia T(n) = T(⌊n/2⌋) + M(n), que se resuelve a T(n) = O(M(n)) vía árboles de recursión. El algoritmo de multiplicación de enteros más rápido conocido (a partir de 2019) funciona en tiempo O(n log n), así que ese es también el tiempo de ejecución del algoritmo más rápido conocido (a partir de 2019) para calcular números de Fibonacci.

¿Es este algoritmo más lento que nuestros algoritmos iterativos de "tiempo lineal"? En realidad, ¡no — la suma tampoco es gratis! Sumar dos números de n dígitos requiere tiempo O(n), así que los algoritmos iterativos IterFibo e IterFibo2 en realidad funcionan en tiempo O(n²). (¿Ves por qué?) Así que FastRecFibo es significativamente más rápido que los algoritmos iterativos, simplemente no exponencialmente más rápido.

En el algoritmo recursivo original, el costo extra de la aritmética de precisión arbitraria es abrumado por el enorme número de llamadas recursivas. La recurrencia correcta es T(n) = T(n − 1) + T(n − 2) + O(n), que todavía tiene la solución T(n) = O(φⁿ).

**3.3 Interpunctio Verborum Redux**

Para nuestro siguiente algoritmo de programación dinámica, consideremos el problema de segmentación de texto del capítulo anterior. Se nos da una cadena A[1 .. n] y una subrutina IsWord que determina si una cadena dada es una palabra (lo que sea que eso signifique), y queremos saber si A puede particionarse en una secuencia de palabras.

Resolvimos este problema definiendo una función Splittable(i) que retorna True si y solo si el sufijo A[i .. n] puede particionarse en una secuencia de palabras. Necesitamos calcular Splittable(1). Esta función satisface la recurrencia

Splittable(i) = { True si i > n ⋁ⱼ₌ᵢⁿ (IsWord(i, j) ∧ Splittable(j + 1)) en caso contrario }

donde IsWord(i, j) es una abreviación para IsWord(A[i .. j]). Esta recurrencia se traduce directamente en un algoritmo recursivo de retroceso que llama a la subrutina IsWord O(2ⁿ) veces en el peor caso.

Pero para cualquier cadena fija A[1 .. n], solo hay n formas diferentes de llamar a la función recursiva Splittable(i) — una para cada valor de i entre 1 y n + 1 — y solo O(n²) formas diferentes de llamar IsWord(i, j) — una para cada par (i, j) tal que 1 ≤ i ≤ j ≤ n. ¿Por qué estamos gastando tiempo exponencial calculando solo una cantidad polinomial de cosas?

Cada subproblema recursivo está especificado por un entero entre 1 y n+1, así que podemos memoizar la función Splittable en un arreglo SplitTable[1 .. n + 1]. Cada subproblema Splittable(i) depende solo de resultados de subproblemas Splittable(j) donde j > i, así que el algoritmo recursivo memoizado llena el arreglo en orden decreciente de índices. Si llenamos el arreglo en este orden deliberadamente, obtenemos el algoritmo de programación dinámica mostrado en la Figura 3.3. El algoritmo hace O(n²) llamadas a IsWord, una mejora exponencial sobre nuestro algoritmo de retroceso anterior.

**Referencia a Figura 3.3**: Interpunctio verborum velox

FastSplittable(A[1 .. n]):

SplitTable[n + 1] ← True

para i ← n hacia abajo hasta 1

SplitTable[i] ← False

para j ← i hasta n

si IsWord(i, j) y SplitTable[j + 1]

SplitTable[i] ← True

retornar SplitTable[1]

**3.4 El Patrón: Recursión Inteligente**

En pocas palabras, la programación dinámica es recursión sin repetición. Los algoritmos de programación dinámica almacenan las soluciones de subproblemas intermedios, a menudo pero no siempre en algún tipo de arreglo o tabla. Muchos estudiantes de algoritmos (e instructores, y libros de texto) cometen el error de enfocarse en la tabla — porque las tablas son fáciles y familiares — en lugar de la tarea mucho más importante (y difícil) de encontrar una recurrencia correcta. Mientras memoicemos la recurrencia correcta, una tabla explícita no es realmente necesaria, pero si la recurrencia es incorrecta, estamos bien y verdaderamente perdidos.

**La programación dinámica no se trata de llenar tablas. ¡Se trata de recursión inteligente!**

Los algoritmos de programación dinámica se desarrollan mejor en dos etapas distintas.

1. **Formular el problema recursivamente.** Escribir una fórmula recursiva o algoritmo para todo el problema en términos de las respuestas a subproblemas más pequeños. Esta es la parte difícil. Una formulación recursiva completa tiene dos partes:

(a) **Especificación.** Describir el problema que quieres resolver recursivamente, en inglés coherente y preciso — no cómo resolver ese problema, sino qué problema estás tratando de resolver. Sin esta especificación, es imposible, incluso en principio, determinar si tu solución es correcta.

(b) **Solución.** Dar una fórmula recursiva clara o algoritmo para todo el problema en términos de las respuestas a instancias más pequeñas del exactamente mismo problema.

1. **Construir soluciones a tu recurrencia de abajo hacia arriba.** Escribir un algoritmo que comience con los casos base de tu recurrencia y se abra camino hacia la solución final, considerando subproblemas intermedios en el orden correcto. Esta etapa puede dividirse en varios pasos más pequeños, relativamente mecánicos:

(a) **Identificar los subproblemas.** ¿Cuáles son todas las formas diferentes en que tu algoritmo recursivo puede llamarse a sí mismo, comenzando con alguna entrada inicial? Por ejemplo, el argumento para RecFibo es siempre un entero entre 0 y n.

(b) **Elegir una estructura de datos de memoización.** Encuentra una estructura de datos que pueda almacenar la solución a cada subproblema que identificaste en el paso (a). Esto es usualmente pero no siempre un arreglo multidimensional.

(c) **Identificar dependencias.** Excepto por los casos base, cada subproblema depende de otros subproblemas — ¿cuáles? Dibuja una imagen de tu estructura de datos, elige un elemento genérico, y dibuja flechas desde cada uno de los otros elementos de los que depende. Luego formaliza tu imagen.

(d) **Encontrar un buen orden de evaluación.** Ordena los subproblemas para que cada uno venga después de los subproblemas de los que depende. Deberías considerar los casos base primero, luego los subproblemas que dependen solo de casos base, y así sucesivamente, eventualmente construyendo hasta el problema original de nivel superior. Las dependencias que identificaste en el paso anterior definen un orden parcial sobre los subproblemas; necesitas encontrar una extensión lineal de ese orden parcial. ¡Ten cuidado!

(e) **Analizar espacio y tiempo de ejecución.** El número de subproblemas distintos determina la complejidad espacial de tu algoritmo memoizado. Para calcular el tiempo total de ejecución, suma los tiempos de ejecución de todos los subproblemas posibles, asumiendo que las llamadas recursivas más profundas ya están memoizadas. En realidad puedes hacer esto inmediatamente después del paso (a).

(f) **Escribir el algoritmo.** Sabes en qué orden considerar los subproblemas, y sabes cómo resolver cada subproblema. ¡Así que hazlo! Si tu estructura de datos es un arreglo, esto usualmente significa escribir algunos bucles for anidados alrededor de tu recurrencia original, y reemplazar las llamadas recursivas con búsquedas en arreglos.

Por supuesto, tienes que probar que cada uno de estos pasos es correcto. Si tu recurrencia está mal, o si tratas de construir respuestas en el orden incorrecto, ¡tu algoritmo no funcionará!

**3.5 Advertencia: La Codicia es Estúpida**

Si tenemos increíble suerte, podemos pasar por alto todas las recurrencias y tablas y demás, y resolver el problema usando un algoritmo codicioso. Como un algoritmo de retroceso, un algoritmo codicioso construye una solución a través de una serie de decisiones, pero toma esas decisiones directamente, sin resolver ningún subproblema recursivo. Aunque este enfoque parece muy natural, casi nunca funciona; los problemas de optimización que pueden resolverse correctamente con un algoritmo codicioso son bastante raros. Sin embargo, para muchos problemas que deberían resolverse con retroceso o programación dinámica, la primera intuición de muchos estudiantes es aplicar una estrategia codiciosa.

Por ejemplo, un algoritmo codicioso para el problema de segmentación de texto podría encontrar el prefijo más corto (o, si prefieres, más largo) de la cadena de entrada que es una palabra, aceptar ese prefijo como la primera palabra en la segmentación, y luego segmentar recursivamente el sufijo restante de la cadena de entrada. De manera similar, un algoritmo codicioso para el problema de la subsecuencia creciente más larga podría buscar el elemento más pequeño del arreglo de entrada, aceptar ese elemento como el inicio de la subsecuencia objetivo, y luego buscar recursivamente la subsecuencia creciente más larga a la derecha de ese elemento. Si estos te suenan como trucos estúpidos, date una palmadita en la espalda; estas ni siquiera están cerca de ser soluciones correctas.

Todos deberían tatuarse la siguiente oración en el dorso de sus manos, justo debajo de todas las reglas sobre logaritmos y notación big-Oh:

**¡Los algoritmos codiciosos nunca funcionan! ¡Usa programación dinámica en su lugar!**

¿Qué, nunca? ¡No, nunca! ¿Qué, nunca? Bueno... casi nunca.¹⁰

Porque el enfoque codicioso es tan increíblemente tentador, pero tan raramente correcto, abogo fuertemente por la siguiente política en cualquier curso de algoritmos, incluso (o quizás especialmente) para cursos que normalmente no piden pruebas de corrección.¹¹

**No recibirás ningún crédito por ningún algoritmo codicioso, en ninguna tarea o examen, incluso si el algoritmo es correcto, sin una prueba formal de corrección.**

Además, la gran mayoría de problemas para los cuales los estudiantes se sienten tentados a enviar un algoritmo codicioso en realidad se resuelven mejor usando programación dinámica. Así que siempre ofrezco el siguiente consejo a mis estudiantes de algoritmos.

**Cuando escribas — o incluso pienses — la palabra "codiciOSO", tu subconsciente te está diciendo que uses programación dinámicA.**

Incluso para problemas que pueden resolverse correctamente con algoritmos codiciosos, usualmente es más productivo desarrollar primero un algoritmo de retroceso o programación dinámica. Primero haz que funcione, luego hazlo rápido. Veremos técnicas para probar que los algoritmos codiciosos son correctos en el siguiente capítulo.

**3.6 Subsecuencia Creciente Más Larga**

Otro problema que consideramos en el capítulo anterior fue calcular la longitud de la subsecuencia creciente más larga de un arreglo dado A[1 .. n] de números. Desarrollamos dos algoritmos recursivos de retroceso diferentes para este problema. Ambos algoritmos funcionan en tiempo O(2ⁿ) en el peor caso; ambos algoritmos pueden acelerarse significativamente vía programación dinámica.

**Primera Recurrencia: ¿Es Este el Siguiente?**

Nuestro primer algoritmo de retroceso evaluó la función LISbigger(i, j), que definimos como la longitud de la subsecuencia creciente más larga de A[j .. n] en la cual cada elemento es mayor que A[i]. Derivamos la siguiente recurrencia para esta función:

LISbigger(i, j) = { 0 si j > n LISbigger(i, j + 1) si A[i] ≥ A[j] max{LISbigger(i, j + 1), 1 + LISbigger(j, j + 1)} en caso contrario }

Para resolver el problema original, podemos agregar un valor centinela A[0] = −∞ al arreglo y calcular LISbigger(0, 1).

Cada subproblema recursivo se identifica por dos índices i y j, así que hay solo O(n²) subproblemas recursivos distintos a considerar. Podemos memoizar los resultados de estos subproblemas en un arreglo bidimensional LISbigger[0 .. n, 1 .. n].¹² Además, cada subproblema puede resolverse en tiempo O(1), sin contar llamadas recursivas, así que deberíamos esperar que el algoritmo final de programación dinámica funcione en tiempo O(n²).

El orden en que el algoritmo recursivo memoizado llena este arreglo no es inmediatamente claro; todo lo que podemos decir de la recurrencia es que cada entrada LISbigger[i, j] se llena después de las entradas LISbigger[i, j+1] y LISbigger[j, j+1] en la siguiente columna, como se indica a la izquierda en la Figura 3.4.

**Referencia a Figura 3.4**: Dependencias de subproblemas para la subsecuencia creciente más larga, y un orden de evaluación válido.

Afortunadamente, esta información parcial es suficiente para darnos un orden de evaluación válido. Si llenamos la tabla una columna a la vez, de derecha a izquierda, entonces cuando alcancemos una entrada en la tabla, las entradas de las que depende ya están disponibles. Este puede no ser el orden que usaría el algoritmo recursivo, pero funciona, así que sigámoslo. La figura derecha en la Figura 3.4 ilustra este orden de evaluación, con una flecha doble indicando el bucle externo y flechas simples indicando el bucle interno. En este caso, las flechas simples son bidireccionales, porque el orden que usamos para llenar cada columna no importa.

¡Y hemos terminado! El pseudocódigo para nuestro algoritmo de programación dinámica se muestra abajo; como se esperaba, nuestro algoritmo claramente funciona en tiempo O(n²). Si es necesario, podemos reducir la cota de espacio de O(n²) a O(n) manteniendo solo las dos columnas más recientes de la tabla, LISbigger[·, j] y LISbigger[·, j + 1].¹³

FastLIS(A[1 .. n]):

A[0] ← −∞ ⟨⟨Agregar un centinela⟩⟩

para i ← 0 hasta n ⟨⟨Casos base⟩⟩

LISbigger[i, n + 1] ← 0

para j ← n hacia abajo hasta 1

para i ← 0 hasta j − 1 ⟨⟨... o lo que sea⟩⟩

keep ← 1 + LISbigger[j, j + 1]

skip ← LISbigger[i, j + 1]

si A[i] ≥ A[j]

LISbigger[i, j] ← skip

sino

LISbigger[i, j] ← max{keep, skip}

retornar LISbigger[0, 1]

**Segunda Recurrencia: ¿Cuál es el Siguiente?**

Nuestro segundo algoritmo de retroceso evaluó la función LISfirst(i), que definimos como la longitud de la subsecuencia creciente más larga de A[i .. n] que comienza con A[i]. Derivamos la siguiente recurrencia para esta función:

LISfirst(i) = 1 + max{LISfirst(j) | j > i y A[j] > A[i]}

Aquí, asumimos que max∅ = 0, así que los casos base como LISfirst(n) = 1 salen de la recurrencia automáticamente. Para resolver el problema original, podemos agregar un valor centinela A[0] = −∞ al arreglo y calcular LISfirst(0) − 1.

En este caso, los subproblemas recursivos se indican por un solo índice i, así que podemos memoizar la recurrencia en un arreglo unidimensional LISfirst[1 .. n]. Cada entrada LISfirst[i] depende solo de entradas LISfirst[j] con j > i, así que podemos llenar el arreglo en orden decreciente de índices. Para calcular cada LISfirst[i], necesitamos considerar LISfirst[j] para todos los índices j > i, pero no necesitamos considerar esos índices j en ningún orden particular. El algoritmo de programación dinámica resultante funciona en tiempo O(n²) y usa espacio O(n).

FastLIS2(A[1 .. n]):

A[0] = −∞ ⟨⟨Agregar un centinela⟩⟩

para i ← n hacia abajo hasta 0

LISfirst[i] ← 1

para j ← i + 1 hasta n ⟨⟨... o lo que sea⟩⟩

si A[j] > A[i] y 1 + LISfirst[j] > LISfirst[i]

LISfirst[i] ← 1 + LISfirst[j]

retornar LISfirst[0] − 1 ⟨⟨No contar el centinela⟩⟩

**3.7 Distancia de Edición**

La distancia de edición entre dos cadenas es el número mínimo de inserciones de letras, eliminaciones de letras y sustituciones de letras requerido para transformar una cadena en la otra. Por ejemplo, la distancia de edición entre FOOD y MONEY es a lo sumo cuatro:

FOOD → MOOD → MOND → MONED → MONEY

Esta función de distancia fue propuesta independientemente por Vladimir Levenshtein en 1965 (trabajando en teoría de codificación), Taras Vintsyuk en 1968 (trabajando en reconocimiento de voz), y Stanislaw Ulam en 1972 (trabajando con secuencias biológicas). Por esta razón, la distancia de edición a veces se llama distancia de Levenshtein o distancia de Ulam (pero extrañamente, nunca "distancia de Vintsyuk").

Podemos visualizar este proceso de edición alineando las cadenas una encima de la otra, con un espacio en la primera palabra por cada inserción y un espacio en la segunda palabra por cada eliminación. Las columnas con dos caracteres diferentes corresponden a sustituciones. En esta representación, el número de pasos de edición es simplemente el número de columnas que no contienen el mismo carácter dos veces.

F O O D -

M O N E Y

Es bastante obvio que no podemos transformar FOOD en MONEY en tres pasos, así que la distancia de edición entre FOOD y MONEY es exactamente cuatro. Desafortunadamente, no es tan fácil en general decir cuándo una secuencia de ediciones es lo más corta posible. Por ejemplo, el siguiente alineamiento muestra que la distancia entre las cadenas ALGORITHM y ALTRUISTIC es a lo sumo 6. ¿Es eso lo mejor que podemos hacer?

A L G O R I T H M -

A L T R U I S T I C

**Estructura Recursiva**

Para desarrollar un algoritmo de programación dinámica para calcular la distancia de edición, primero necesitamos formular el problema recursivamente. Nuestra representación de alineamiento para secuencias de edición tiene una propiedad crucial de "estructura óptima". Supón que tenemos la representación de espacios para la secuencia de edición más corta para dos cadenas. Si eliminamos la última columna, las columnas restantes deben representar la secuencia de edición más corta para los prefijos restantes. Podemos probar fácilmente esta observación por contradicción: Si los prefijos tuvieran una secuencia de edición más corta, pegar la última columna nos daría una secuencia de edición más corta para las cadenas originales.

Dicho de manera diferente, el alineamiento que estamos buscando representa una secuencia de operaciones de edición, ordenadas (sin razón particular) de derecha a izquierda. Resolver el problema de distancia de edición requiere tomar una secuencia de decisiones, una para cada columna en el alineamiento de salida. En el medio de esta secuencia de decisiones, ya hemos alineado un sufijo de una cadena con un sufijo de la otra.

ALGOR ITHM

ALTRU ISTIC

Porque el costo de un alineamiento es simplemente el número de columnas no coincidentes, nuestras decisiones restantes no dependen de las operaciones de edición que ya hemos elegido; solo dependen de los prefijos que aún no hemos alineado.

ALGOR

ALTRU

Así, para cualquiera dos cadenas de entrada A[1 .. m] y B[1 .. n], podemos formular el problema de distancia de edición recursivamente de la siguiente manera: Para cualquier índice i y j, sea Edit(i, j) la distancia de edición entre los prefijos A[1 .. i] y B[1 .. j]. Necesitamos calcular Edit(m, n).

**Recurrencia**

Cuando i y j son ambos positivos, hay exactamente tres posibilidades para la última columna en el alineamiento óptimo de A[1 .. i] y B[1 .. j]:

• **Inserción**: La última entrada en la fila superior está vacía. En este caso, la distancia de edición es igual a Edit(i, j −1) +1. El +1 es el costo de la inserción final, y la expresión recursiva da el costo mínimo para el alineamiento restante.

ALGOR

ALTRU

• **Eliminación**: La última entrada en la fila inferior está vacía. En este caso, la distancia de edición es igual a Edit(i − 1, j) + 1. El +1 es el costo de la eliminación final, y la expresión recursiva da el costo mínimo para el alineamiento restante.

ALGO R

ALTRU

• **Sustitución**: Ambas filas tienen caracteres en la última columna. Si estos dos caracteres son diferentes, entonces la distancia de edición es igual a Edit(i−1, j−1)+1. Si estos dos caracteres son iguales, la sustitución es gratis, así que la distancia de edición es Edit(i − 1, j − 1).

ALGO R ALGO R

ALTR U ALT R

Este análisis de casos genérico se descompone si i = 0 o j = 0, pero esos casos límite son fáciles de manejar directamente.

• Transformar la cadena vacía en una cadena de longitud j requiere j inserciones, así que Edit(0, j) = j.

• Transformar una cadena de longitud i en la cadena vacía requiere i eliminaciones, así que Edit(i, 0) = i.

Como una verificación de cordura, ¡ambos casos base indican correctamente que la distancia de edición entre la cadena vacía y la cadena vacía es cero!

Concluimos que la función Edit satisface la siguiente recurrencia:

Edit(i, j) = { i si j = 0 j si i = 0 min{Edit(i, j − 1) + 1, Edit(i − 1, j) + 1, Edit(i − 1, j − 1) + [A[i] ≠ B[j]]} en caso contrario }

**Programación Dinámica**

Ahora que tenemos una recurrencia, podemos transformarla en un algoritmo de programación dinámica siguiendo nuestra receta mecánica usual.

• **Subproblemas**: Cada subproblema recursivo se identifica por dos índices 0 ≤ i ≤ m y 0 ≤ j ≤ n.

• **Estructura de memoización**: Así que podemos memoizar todos los valores posibles de Edit(i, j) en un arreglo bidimensional Edit[0 .. m, 0 .. n].

• **Dependencias**: Cada entrada Edit[i, j] depende solo de sus tres entradas vecinas Edit[i − 1, j], Edit[i, j − 1], y Edit[i − 1, j − 1].

• **Orden de evaluación**: Si llenamos este arreglo en orden estándar fila-mayor — fila por fila de arriba hacia abajo, cada fila de izquierda a derecha — entonces cuando alcancemos una entrada en el arreglo, todas las entradas de las que depende ya están disponibles. (Este no es el único orden de evaluación que podríamos usar, pero funciona, así que sigámoslo.)

j →

i ↓ [diagram showing dependency pattern]

• **Espacio y tiempo**: La estructura de memoización usa espacio O(mn). Podemos calcular cada entrada Edit[i, j] en tiempo O(1) una vez que conocemos sus predecesores, así que el algoritmo general funciona en tiempo O(mn).

Aquí está el algoritmo de programación dinámica resultante:

EditDistance(A[1 .. m], B[1 .. n]):

para j ← 0 hasta n

Edit[0, j] ← j

para i ← 1 hasta m

Edit[i, 0] ← i

para j ← 1 hasta n

ins ← Edit[i, j − 1] + 1

del ← Edit[i − 1, j] + 1

si A[i] = B[j]

rep ← Edit[i − 1, j − 1]

sino

rep ← Edit[i − 1, j − 1] + 1

Edit[i, j] ← min{ins, del, rep}

retornar Edit[m, n]

Este algoritmo se atribuye más comúnmente a Robert Wagner y Michael Fischer, quienes describieron el algoritmo en 1974. Sin embargo, en pleno cumplimiento con la Ley de Eponimia de Stigler, algoritmos idénticos o más generales fueron descubiertos independientemente por Taras Vintsyuk en 1968, V. M. Velichko y N. G. Zagoruyko en 1970, David Sankoff en 1972, Peter Sellers en 1974, y casi ciertamente varios otros.¹⁴ Curiosamente, ¡ninguno de estos autores cita ni a Levenshtein ni a Ulam!

La tabla de memoización para las cadenas de entrada ALGORITHM y ALTRUISTIC se muestra abajo. Los números en negrita indican lugares donde los caracteres en las dos cadenas son iguales. ¡La distancia de edición entre ALGORITHM y ALTRUISTIC es de hecho seis!

A L G O R I T H M

0→1→2→3→4→5→6→7→8→9

A 1→0→1→2→3→4→5→6→7→8

L 2→1→0→1→2→3→4→5→6→7

T 3→2→1→1→2→3→4→4→5→6

R 4→3→2→2→2→2→3→4→5→6

U 5→4→3→3→3→3→3→4→5→6

I 6→5→4→4→4→4→3→4→5→6

S 7→6→5→5→5→5→4→4→5→6

T 8→7→6→6→6→6→5→4→5→6

I 9→8→7→7→7→7→6→5→5→6

C 10→9→8→8→8→8→7→6→6→6

Las flechas en esta tabla indican cuál(es) predecesor(es) realmente definen cada entrada. Cada dirección de flecha corresponde a una operación de edición diferente: horizontal=eliminación, vertical=inserción, y diagonal=sustitución. Las flechas diagonales rojas en negrita indican sustituciones "gratuitas" de una letra por sí misma. Cualquier camino de flechas desde la esquina superior izquierda hasta la esquina inferior derecha de esta tabla representa una secuencia óptima de edición entre las dos cadenas. El arreglo de memoización de ejemplo contiene exactamente tres caminos dirigidos desde la esquina superior izquierda hasta la esquina inferior derecha, cada uno indicando una secuencia diferente de seis ediciones transformando ALGORITHM en ALTRUISTIC, como se muestra en la página siguiente.

A L G O R I T H M

A L T R U I S T I C

A L G O R I T H M

A L T R U I S T I C

A L G O R I T H M

A L T R U I S T I C

Nuestro algoritmo EditDistance en realidad no calcula o almacena ninguna flecha en la tabla, pero la(s) flecha(s) que llevan a cualquier entrada en la tabla pueden reconstruirse al vuelo en tiempo O(1) a partir de los valores numéricos. Así, una vez que hemos llenado la tabla, podemos reconstruir la secuencia de edición más corta en tiempo adicional O(n+m).

**3.8 Suma de Subconjuntos**

Recuerda que el problema de Suma de Subconjuntos pregunta si algún subconjunto de un arreglo dado X[1 .. n] de enteros positivos suma a un entero dado T. En el capítulo anterior, desarrollamos un algoritmo recursivo de Suma de Subconjuntos que puede reformularse de la siguiente manera. Fija el arreglo de entrada original X[1 .. n] y define la función booleana SS(i, t) = True si y solo si algún subconjunto de X[i .. n] suma a t.

Necesitamos calcular SS(1, T). Esta función satisface la siguiente recurrencia:

SS(i, t) = { True si t = 0 False si t < 0 o i > n SS(i + 1, t) ∨ SS(i + 1, t − X[i]) en caso contrario }

Podemos transformar esta recurrencia en un algoritmo de programación dinámica siguiendo el texto estándar usual.

• **Subproblemas**: Cada subproblema se describe por un entero i tal que 1 ≤ i ≤ n + 1, y un entero t ≤ T. Sin embargo, los subproblemas con t < 0 son triviales, así que parece bastante tonto memoizarlos.¹⁵ De hecho, podemos modificar la recurrencia para que esos subproblemas nunca surjan:

SS(i, t) = { True si t = 0 False si i > n SS(i + 1, t) si t < X[i] SS(i + 1, t) ∨ SS(i + 1, t − X[i]) en caso contrario }

• **Estructura de datos**: Podemos memoizar nuestra recurrencia en un arreglo bidimensional S[1 .. n + 1, 0 .. T], donde S[i, t] almacena el valor de SS(i, t).

• **Orden de evaluación**: Cada entrada S[i, t] depende de a lo sumo dos otras entradas, ambas de la forma SS[i + 1,·]. Así que podemos llenar el arreglo considerando filas de abajo hacia arriba en el bucle externo, y considerando los elementos en cada fila en orden arbitrario en el bucle interno.

• **Espacio y tiempo**: La estructura de memoización usa espacio O(nT). Si S[i+1, t] y S[i +1, t − X[i]] ya se conocen, podemos calcular S[i, t] en tiempo constante, así que el algoritmo funciona en tiempo O(nT).

Aquí está el algoritmo de programación dinámica resultante:

FastSubsetSum(X[1 .. n], T):

S[n + 1, 0] ← True

para t ← 1 hasta T

S[n + 1, t] ← False

para i ← n hacia abajo hasta 1

S[i, 0] = True

para t ← 1 hasta X[i] − 1

S[i, t] ← S[i + 1, t] ⟨⟨Evitar el caso t < 0⟩⟩

para t ← X[i] hasta T

S[i, t] ← S[i + 1, t] ∨ S[i + 1, t − X[i]]

retornar S[1, T]

El tiempo de ejecución en el peor caso O(nT) para este algoritmo es una mejora significativa sobre el algoritmo recursivo de retroceso de tiempo O(2ⁿ) cuando T es pequeño.¹⁶ Sin embargo, si la suma objetivo T es significativamente mayor que 2ⁿ, este algoritmo iterativo es en realidad más lento que el algoritmo recursivo ingenuo, porque está perdiendo tiempo resolviendo subproblemas que el algoritmo recursivo nunca considera. ¡La programación dinámica no siempre es una mejora!¹⁷

**3.9 Árboles de Búsqueda Binaria Óptimos**

El problema final que consideramos en el capítulo anterior fue el problema del árbol de búsqueda binaria óptimo. La entrada es un arreglo ordenado A[1 .. n] de claves de búsqueda y un arreglo f [1 .. n] de conteos de frecuencia, donde f [i] es el número de veces que buscaremos A[i]. Nuestra tarea es construir un árbol de búsqueda binaria para ese conjunto tal que el costo total de todas las búsquedas sea lo más pequeño posible.

Fija el arreglo de frecuencias f, y sea OptCost(i, k) el tiempo total de búsqueda en el árbol de búsqueda óptimo para el subarreglo A[i .. k]. Derivamos la siguiente recurrencia para la función OptCost:

OptCost(i, k) = { 0 si i > k Σⱼ₌ᵢᵏ f[j] + min\_{i≤r≤k} (OptCost(i,r − 1) + OptCost(r + 1, k)) en caso contrario }

Probablemente puedas adivinar lo que vamos a hacer con esta recurrencia eventualmente, pero deshagámonos de esa suma fea primero.

Para cualquier par de índices i ≤ k, sea F(i, k) el conteo total de frecuencia para todas las claves en el intervalo A[i .. k]:

F(i, k) := Σⱼ₌ᵢᵏ f[j]

Esta función satisface la siguiente recurrencia simple:

F(i, k) = { f[i] si i = k F(i, k − 1) + f[k] en caso contrario }

Podemos calcular todos los valores posibles de F(i, k) en tiempo O(n²) usando — ¡lo adivinaste! — programación dinámica. Los pasos mecánicos usuales nos dan el siguiente algoritmo de programación dinámica:

InitF(f [1 .. n]):

para i ← 1 hasta n

F[i, i − 1] ← 0

para k ← i hasta n

F[i, k] ← F[i, k − 1] + f[k]

Usaremos este algoritmo corto como una subrutina de inicialización. Esta inicialización nos permite simplificar la recurrencia OptCost original de la siguiente manera:

OptCost(i, k) = { 0 si i > k F[i, k] + min\_{i≤r≤k} (OptCost(i,r − 1) + OptCost(r + 1, k)) en caso contrario }

Ahora hagamos girar la manivela.

• **Subproblemas**: Cada subproblema recursivo se especifica por dos enteros i y k, tales que 1 ≤ i ≤ n + 1 y 0 ≤ k ≤ n.

• **Memoización**: Podemos almacenar todos los valores posibles de OptCost en un arreglo bidimensional OptCost[1 .. n + 1, 0 .. n]. (Solo las entradas OptCost[i, j] con j ≥ i − 1 serán realmente usadas, pero lo que sea.)

• **Dependencias**: Cada entrada OptCost[i, k] depende de las entradas OptCost[i, j − 1] y OptCost[j + 1, k], para toda j tal que i ≤ j ≤ k. En otras palabras, cada entrada de la tabla depende de todas las entradas ya sea directamente a la izquierda o directamente abajo.

k →

i ↓ [dependency diagram]

La siguiente subrutina llena la entrada OptCost[i, k], asumiendo que todas las entradas de las que depende ya han sido calculadas.

ComputeOptCost(i, k):

OptCost[i, k] ← ∞

para r ← i hasta k

tmp ← OptCost[i,r − 1] + OptCost[r + 1, k]

si OptCost[i, k] > tmp

OptCost[i, k] ← tmp

OptCost[i, k] ← OptCost[i, k] + F[i, k]

• **Orden de evaluación**: Hay al menos tres órdenes diferentes que pueden usarse para llenar el arreglo. El primero que se le ocurre a la mayoría de estudiantes es escanear a través de la tabla una diagonal a la vez, comenzando con los casos base triviales OptCost[i, i − 1] y trabajando hacia la respuesta final OptCost[1, n], así:

OptimalBST(f [1 .. n]):

InitF(f [1 .. n])

para i ← 1 hasta n + 1

OptCost[i, i − 1] ← 0

para d ← 0 hasta n − 1

para i ← 1 hasta n − d ⟨⟨... o lo que sea⟩⟩

ComputeOptCost(i, i + d)

retornar OptCost[1, n]

También podríamos atravesar el arreglo fila por fila de abajo hacia arriba, atravesando cada fila de izquierda a derecha, o columna por columna de izquierda a derecha, atravesando cada columna de abajo hacia arriba.

OptimalBST2(f [1 .. n]):

InitF(f [1 .. n])

para i ← n + 1 hacia abajo hasta 1

OptCost[i, i − 1] ← 0

para j ← i hasta n

ComputeOptCost(i, j)

retornar OptCost[1, n]

OptimalBST3(f [1 .. n]):

InitF(f [1 .. n])

para j ← 0 hasta n + 1

OptCost[j + 1, j] ← 0

para i ← j hacia abajo hasta 1

ComputeOptCost(i, j)

retornar OptCost[1, n]

Como antes, podemos ilustrar estos órdenes de evaluación usando una flecha de línea doble para indicar el bucle externo y flechas de línea simple para indicar el bucle interno. Las flechas bidireccionales en el primer orden de evaluación indican que el orden de los bucles internos no importa.

• **Tiempo y espacio**: La estructura de memoización usa espacio O(n²). Sin importar qué orden de evaluación elijamos, necesitamos tiempo O(n) para calcular cada entrada OptCost[i, k], así que nuestro algoritmo general funciona en tiempo O(n³).

Como de costumbre, podríamos haber predicho las cotas finales de espacio y tiempo directamente de la recurrencia original:

OptCost(i, k) = { 0 si i > k F[i, k] + min\_{i≤r≤k} (OptCost(i,r − 1) + OptCost(r + 1, k)) en caso contrario }

La función OptCost tiene dos argumentos, cada uno de los cuales puede tomar aproximadamente n valores diferentes, así que probablemente necesitamos una estructura de datos de tamaño O(n²). Por otro lado, hay tres variables en el cuerpo de la recurrencia (i, k, y r), cada una de las cuales puede tomar aproximadamente n valores diferentes, así que debería tomar tiempo O(n³) calcular todo.

**3.10 Programación Dinámica en Árboles**

Hasta ahora, todos nuestros ejemplos de programación dinámica usan arreglos multidimensionales para almacenar los resultados de subproblemas recursivos. Sin embargo, como muestra el siguiente ejemplo, esta no siempre es la estructura de datos más apropiada para usar.

Un conjunto independiente en un grafo es un subconjunto de los vértices sin aristas entre ellos. Encontrar el conjunto independiente más grande en un grafo arbitrario es extremadamente difícil; de hecho, este es uno de los problemas NP-difíciles canónicos que estudiaremos en el Capítulo 12. Pero en algunas clases especiales de grafos, podemos encontrar conjuntos independientes más grandes rápidamente. En particular, cuando el grafo de entrada es un árbol con n vértices, en realidad podemos calcular el conjunto independiente más grande en tiempo O(n).

Supón que se nos da un árbol T. Sin pérdida de generalidad, supón que T es un árbol con raíz; es decir, hay un nodo especial en T llamado la raíz, y todas las aristas están implícitamente dirigidas alejándose de este vértice. (Si T es un árbol sin raíz — un grafo no dirigido acíclico conectado — podemos elegir un vértice arbitrario como la raíz.) Llamamos al vértice w un descendiente del vértice v si el camino único de w a la raíz incluye a v; equivalentemente, los descendientes de v son v mismo y los descendientes de los hijos de v. El subárbol con raíz en v consiste en todos los descendientes de v y las aristas entre ellos.

Para cualquier nodo v en T, sea MIS(v) el tamaño del conjunto independiente más grande en el subárbol con raíz en v. Cualquier conjunto independiente en este subárbol que excluye a v mismo es la unión de conjuntos independientes en los subárboles con raíz en los hijos de v. Por otro lado, cualquier conjunto independiente que incluye a v necesariamente excluye a todos los hijos de v, y por lo tanto incluye conjuntos independientes en los subárboles con raíz en los nietos de v. Así, la función MIS obedece la siguiente recurrencia, donde la notación no estándar w ↓ v significa "w es un hijo de v":

MIS(v) = max(Σ\_{w↓v} MIS(w), 1 + Σ\_{w↓v} Σ\_{x↓w} MIS(x))

Necesitamos calcular MIS(r), donde r es la raíz de T.

**Referencia a Figura 3.5**: Calcular el conjunto independiente máximo en un árbol

¿Qué estructura de datos deberíamos usar para memoizar esta recurrencia? ¡La elección más natural es el árbol T mismo! Específicamente, para cada vértice v en T, almacenamos el resultado de MIS(v) en un nuevo campo v.MIS. (En principio, podríamos usar un arreglo en su lugar, pero entonces necesitaríamos punteros de ida y vuelta entre cada nodo y su entrada de arreglo correspondiente, ¿así que para qué molestarse?)

¿Cuál es un buen orden para considerar los subproblemas? El subproblema asociado con cualquier nodo v depende de los subproblemas asociados con los hijos y nietos de v. Así que podemos visitar los nodos en cualquier orden que nos guste, siempre que cada vértice sea visitado antes que su padre; en particular, podemos usar un recorrido post-orden estándar.

¿Cuál es el tiempo de ejecución del algoritmo? El tiempo no recursivo asociado con cada nodo v es proporcional al número de hijos y nietos de v; este número puede ser muy diferente de un vértice al siguiente. Pero podemos dar vuelta al análisis: ¡Cada vértice contribuye una cantidad constante de tiempo a su padre y su abuelo! Porque cada vértice tiene a lo sumo un padre y a lo sumo un abuelo, el algoritmo funciona en tiempo O(n).

Aquí está el algoritmo de programación dinámica resultante. Sí, todavía es recursivo, porque esa es la forma más natural de implementar un recorrido post-orden del árbol.

TreeMIS(v):

skipv ← 0

para cada hijo w de v

skipv ← skipv + TreeMIS(w)

keepv ← 1

para cada nieto x de v

keepv ← keepv + x.MIS

v.MIS ← max{keepv, skipv}

retornar v.MIS

Podemos derivar un algoritmo lineal aún más simple definiendo dos funciones separadas sobre los nodos de T:

• Sea MISyes(v) el tamaño del conjunto independiente más grande del subárbol con raíz en v que incluye a v.

• Sea MISno(v) el tamaño del conjunto independiente más grande del subárbol con raíz en v que excluye a v.

De nuevo, necesitamos calcular max{MISyes(r), MISno(r)}, donde r es la raíz de T.

Las primeras dos funciones satisfacen la siguiente recurrencia mutua:

MISyes(v) = 1 + Σ\_{w↓v} MISno(w)

MISno(v) = Σ\_{w↓v} max{MISyes(w), MISno(w)}

De nuevo, podemos memoizar estas funciones en el árbol mismo, definiendo dos nuevos campos para cada vértice. Un recorrido post-orden sencillo del árbol evalúa ambas funciones en cada nodo en tiempo O(n). El siguiente algoritmo no solo memoiza ambos valores de función en v, sino que también retorna el mayor de esos dos valores.

TreeMIS2(v):

v.MISno ← 0

v.MISyes ← 1

para cada hijo w de v

v.MISno ← v.MISno + TreeMIS2(w)

v.MISyes ← v.MISyes + w.MISno

retornar max{v.MISyes, v.MISno}

En la segunda línea del bucle interno, estamos usando el valor w.MISno que fue memoizado por la llamada recursiva en la línea anterior.

**Ejercicios del Capítulo 3 - Programación Dinámica**

Para todos los siguientes ejercicios — y más generalmente al desarrollar cualquier nuevo algoritmo de programación dinámica — recomiendo fuertemente seguir los pasos delineados en la Sección 3.4. En particular, ni siquiera comiences a pensar en tablas o bucles for hasta que tengas una solución recursiva completa, incluyendo una especificación clara en inglés de los subproblemas recursivos que realmente estás resolviendo.¹⁸

**Primero haz que funcione, luego hazlo rápido.**

**Secuencias/Arreglos**

**1.** En una vida anterior, trabajaste como cajero en la perdida colonia antártica de Nadiria, pasando la mayor parte de tu día dando cambio a tus clientes. Porque el papel es un recurso muy raro y valioso en la Antártida, los cajeros estaban obligados por ley a usar la menor cantidad posible de billetes cuando daban cambio. Gracias a las predilecciones numerológicas de uno de sus fundadores, la moneda de Nadiria, llamada Dream-Dollars, estaba disponible en las siguientes denominaciones: $1, $4, $7, $13, $28, $52, $91, y $365.¹⁹

♠ (a) El algoritmo codicioso de cambio repetidamente toma el billete más grande que no excede la cantidad objetivo. Por ejemplo, para hacer $122 usando el algoritmo codicioso, primero tomamos un billete de $91, luego un billete de $28, y finalmente tres billetes de $1. Da un ejemplo donde este algoritmo codicioso usa más billetes Dream-Dollar que el mínimo posible. [Pista: Puede ser más fácil escribir un pequeño programa que calcularlo a mano.]

(b) Describe y analiza un algoritmo recursivo que calcule, dado un entero k, el número mínimo de billetes necesarios para hacer k Dream-Dollars. (No te preocupes por hacer tu algoritmo rápido; solo asegúrate de que sea correcto.)

(c) Describe un algoritmo de programación dinámica que calcule, dado un entero k, el número mínimo de billetes necesarios para hacer k Dream-Dollars. (Este debe ser rápido.)

**2.** Describe algoritmos eficientes para las siguientes variantes del problema de segmentación de texto. Asume que tienes una subrutina IsWord que toma un arreglo de caracteres como entrada y retorna True si y solo si esa cadena es una "palabra". Analiza tus algoritmos acotando el número de llamadas a IsWord.

(a) Dado un arreglo A[1 .. n] de caracteres, calcula el número de particiones de A en palabras. Por ejemplo, dada la cadena ARTISTOIL, tu algoritmo debería retornar 2, para las particiones ARTIST·OIL y ART·IS·TOIL.

(b) Dados dos arreglos A[1 .. n] y B[1 .. n] de caracteres, decide si A y B pueden particionarse en palabras en los mismos índices. Por ejemplo, las cadenas BOTHEARTHANDSATURNSPIN y PINSTARTRAPSANDRAGSLAP pueden particionarse en palabras en los mismos índices de la siguiente manera:

BOT·HEART·HAND·SAT·URNS·PIN

PIN·START·RAPS·AND·RAGS·LAP

(c) Dados dos arreglos A[1 .. n] y B[1 .. n] de caracteres, calcula el número de diferentes formas en que A y B pueden particionarse en palabras en los mismos índices.

**3.** Supón que se te da un arreglo A[1 .. n] de números, que pueden ser positivos, negativos, o cero, y que no son necesariamente enteros.

(a) Describe y analiza un algoritmo que encuentra la suma más grande de elementos en un subarreglo contiguo A[i .. j].

(b) Describe y analiza un algoritmo que encuentra el producto más grande de elementos en un subarreglo contiguo A[i .. j].

Por ejemplo, dado el arreglo [−6, 12, −7, 0, 14, −7, 5] como entrada, tu primer algoritmo debería retornar 19, y tu segundo algoritmo debería retornar 504.

suma=19

⎫

−6 12 −7 0 14 −7 5

⎭‾‾‾‾‾‾‾‾‾‾‾⎫

producto=504

Dado el arreglo de un elemento [−374] como entrada, tu primer algoritmo debería retornar 0, y tu segundo algoritmo debería retornar 1. (¡El intervalo vacío todavía es un intervalo!) Para propósitos de análisis, asume que comparar, sumar, o multiplicar cualquier par de números toma tiempo O(1).

[Pista: La parte (a) ha sido una pregunta estándar de entrevista de ciencias de la computación desde al menos mediados de los años 1980. Puedes encontrar muchas soluciones correctas en la web; ¡el problema incluso tiene su propia página de Wikipedia! Pero al menos en 2016, una fracción significativa de las soluciones que encontré en la web para la parte (b) eran ya sea más lentas de lo necesario o en realidad incorrectas.]

**4.** Este ejercicio explora variantes del problema del subarreglo máximo (Problema 3). En todos los casos, tu entrada consiste en un arreglo A[1 .. n] de números reales (que podrían ser positivos, negativos, o cero) y posiblemente un entero adicional X ≥ 0.

(a) **Envolvente**: Supón que A es un arreglo circular. En este contexto, un "subarreglo contiguo" puede ser ya sea un intervalo A[i .. j] o un sufijo seguido por un prefijo A[i .. n] · A[1 .. j]. Describe y analiza un algoritmo que encuentra un subarreglo contiguo de A con la suma más grande.

(b) **Solo subarreglos largos**: Describe y analiza un algoritmo que encuentra un subarreglo contiguo de A de longitud al menos X que tiene la suma más grande. (Asume X ≤ n.)

(c) **Solo subarreglos cortos**: Describe y analiza un algoritmo que encuentra un subarreglo contiguo de A de longitud a lo sumo X que tiene la suma más grande.

(d) **El Precio es Correcto**: Describe y analiza un algoritmo que encuentra un subarreglo contiguo de A con la suma más grande menor o igual a X.

(e) Describe un algoritmo más rápido para el Problema 4(d) cuando cada número en el arreglo A es no negativo.

**5.** Este ejercicio te pide desarrollar algoritmos eficientes para encontrar subsecuencias óptimas de varios tipos. Una subsecuencia es cualquier cosa obtenida de una secuencia extrayendo un subconjunto de elementos, pero manteniéndolos en el mismo orden; los elementos de la subsecuencia no necesitan ser contiguos en la secuencia original. Por ejemplo, las cadenas C, DAMN, YAIOAI, y DYNAMICPROGRAMMING son todas subsecuencias de la cadena DYNAMICPROGRAMMING.

[Pista: Exactamente uno de estos problemas puede resolverse en tiempo O(n) usando un algoritmo codicioso.]

(a) Sean A[1 .. m] y B[1 .. n] dos arreglos arbitrarios. Una subsecuencia común de A y B es otra secuencia que es subsecuencia de tanto A como B. Describe un algoritmo eficiente para calcular la longitud de la subsecuencia común más larga de A y B.

(b) Sean A[1 .. m] y B[1 .. n] dos arreglos arbitrarios. Una supersecuencia común de A y B es otra secuencia que contiene tanto A como B como subsecuencias. Describe un algoritmo eficiente para calcular la longitud de la supersecuencia común más corta de A y B.

(c) Llama a una secuencia X[1 .. n] de números **bitónica** si hay un índice i con 1 < i < n, tal que el prefijo X[1 .. i] es creciente y el sufijo X[i .. n] es decreciente. Describe un algoritmo eficiente para calcular la longitud de la subsecuencia bitónica más larga de un arreglo arbitrario A de enteros.

(d) Llama a una secuencia X[1 .. n] de números **oscilante** si X[i] < X[i + 1] para todo i par, y X[i] > X[i + 1] para todo i impar. Describe un algoritmo eficiente para calcular la longitud de la subsecuencia oscilante más larga de un arreglo arbitrario A de enteros.

(e) Describe un algoritmo eficiente para calcular la longitud de la supersecuencia oscilante más corta de un arreglo arbitrario A de enteros.

(f) Llama a una secuencia X[1 .. n] de números **convexa** si 2·X[i] < X[i−1]+X[i+1] para todo i. Describe un algoritmo eficiente para calcular la longitud de la subsecuencia convexa más larga de un arreglo arbitrario A de enteros.

(g) Llama a una secuencia X[1 .. n] de números **débilmente creciente** si cada elemento es mayor que el promedio de los dos elementos anteriores; es decir, 2·X[i] > X[i − 1] + X[i − 2] para todo i > 2. Describe un algoritmo eficiente para calcular la longitud de la subsecuencia débilmente creciente más larga de un arreglo arbitrario A de enteros.

(h) Llama a una secuencia X[1 .. n] de números **doble-creciente** si X[i] > X[i−2] para todo i > 2. (En otras palabras, una secuencia doble-creciente es una mezcla perfecta de dos secuencias crecientes.) Describe un algoritmo eficiente para calcular la longitud de la subsecuencia doble-creciente más larga de un arreglo arbitrario A de enteros.

(i) Recuerda que una secuencia X[1 .. n] de números es **creciente** si X[i] < X[i+1] para todo i. Describe un algoritmo eficiente para calcular la longitud de la subsecuencia común creciente más larga de dos arreglos dados de enteros. Por ejemplo, ⟨1, 4, 5, 6, 7, 9⟩ es la subsecuencia común creciente más larga de las secuencias ⟨3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3⟩ y ⟨1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5⟩.

**6.** Una **mezcla** de dos cadenas X e Y se forma intercalando los caracteres en una nueva cadena, manteniendo los caracteres de X e Y en el mismo orden. Por ejemplo, la cadena BANANAANANAS es una mezcla de las cadenas BANANA y ANANAS de varias maneras diferentes.

BANANAANANAS BANANAANANAS BANANAANANAS

De manera similar, las cadenas PRODGYRNAMAMMIINCG y DYPRONGARMAMMICING son ambas mezclas de DYNAMIC y PROGRAMMING:

PRODGYRNAMAMMIINCG DYPRONGARMAMMICING

(a) Dadas tres cadenas A[1 .. m], B[1 .. n], y C[1 .. m + n], describe y analiza un algoritmo para determinar si C es una mezcla de A y B.

(b) Una **mezcla suave** de X e Y es una mezcla de X e Y que nunca usa más de dos símbolos consecutivos de cualquier cadena. Por ejemplo,

• PRDOYGNARAMMMIICNG es una mezcla suave de las cadenas DYNAMIC y PROGRAMMING. • DYPRNOGRAAMMMICING es una mezcla de DYNAMIC y PROGRAMMING, pero no es una mezcla suave (debido a las subcadenas OGR e ING). • XXXXXXXXXXXXXXXXXXX es una mezcla suave de las cadenas XXXXXXX y XXXXXXXXXXX. • No hay mezcla suave de las cadenas XXXX y XXXXXXXXXXXX.

Describe y analiza un algoritmo para decidir, dadas tres cadenas X, Y, y Z, si Z es una mezcla suave de X e Y.

**7.** Para cada uno de los siguientes problemas, la entrada consiste en dos arreglos X[1 .. k] e Y[1 .. n] donde k ≤ n.

(a) Describe y analiza un algoritmo para decidir si X es una subsecuencia de Y. Por ejemplo, la cadena PPAP es una subsecuencia de la cadena PENPINEAPPLEAPPLEPEN.

(b) Describe y analiza un algoritmo para encontrar el menor número de símbolos que pueden eliminarse de Y para que X ya no sea una subsecuencia. Equivalentemente, tu algoritmo debería encontrar la subsecuencia más larga de Y que no es una supersecuencia de X. Por ejemplo, después de eliminar dos símbolos de la cadena PENPINEAPPLEAPPLEPEN, la cadena PPAP ya no es una subsecuencia.

♥(c) Describe y analiza un algoritmo para determinar si X ocurre como dos subsecuencias disjuntas de Y. Por ejemplo, la cadena PPAP aparece como dos subsecuencias disjuntas en la cadena PENPINEAPPLEAPPLEPEN.

(d) Supón que la entrada también incluye un tercer arreglo C[1 .. n] de números, que pueden ser positivos, negativos, o cero, donde C[i] es el costo de Y[i]. Describe y analiza un algoritmo para calcular la ocurrencia de costo mínimo de X como una subsecuencia de Y. Es decir, queremos encontrar un arreglo I[1 .. k] tal que I[j] < I[j + 1] y X[I[j]] = Y[j] para cada índice j, y el costo total ∑ⱼ₌₁ᵏ C[j] sea lo más pequeño posible.

(e) Describe y analiza un algoritmo para calcular el número total de ocurrencias (posiblemente superpuestas) de X como una subsecuencia de Y. Para propósitos de análisis, asume que podemos sumar dos enteros arbitrarios en tiempo O(1). Por ejemplo, la cadena PPAP aparece exactamente 23 veces como una subsecuencia de la cadena PENPINEAPPLEAPPLEPEN. Si todos los caracteres en X e Y son iguales, tu algoritmo debería retornar (n choose k).

(f) ¿Cuál es el tiempo de ejecución de tu algoritmo para la parte (d) si sumar dos enteros de ℓ bits requiere tiempo O(ℓ)?

**8.** Describe y analiza un algoritmo eficiente para encontrar la longitud de la subcadena contigua más larga que aparece tanto hacia adelante como hacia atrás en una cadena de entrada T[1 .. n]. Las subcadenas hacia adelante y hacia atrás no deben superponerse. Aquí hay varios ejemplos:

• Dada la cadena de entrada ALGORITHM, tu algoritmo debería retornar 0. • Dada la cadena de entrada RECURSION, tu algoritmo debería retornar 1, para la subcadena R. • Dada la cadena de entrada REDIVIDE, tu algoritmo debería retornar 3, para la subcadena EDI. (¡Las subcadenas hacia adelante y hacia atrás no deben superponerse!) • Dada la cadena de entrada DYNAMICPROGRAMMINGMANYTIMES, tu algoritmo debería retornar 4, para la subcadena YNAM. (En particular, no debería retornar 6, para la subsecuencia YNAMIR).

**9.** Un **palíndromo** es cualquier cadena que es exactamente igual a su reverso, como I, o DEED, o RACECAR, o AMANAPLANACATACANALPANAMA.

(a) Describe y analiza un algoritmo para encontrar la longitud de la subsecuencia más larga de una cadena dada que también es un palíndromo.

Por ejemplo, la subsecuencia palindrómica más larga de la cadena MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM es MHYMRORMYHM; así, dada esa cadena como entrada, tu algoritmo debería retornar 11.

(b) Describe y analiza un algoritmo para encontrar la longitud de la supersecuencia más corta de una cadena dada que también es un palíndromo. Por ejemplo, la supersecuencia palindrómica más corta de TWENTYONE es TWENTOYOTNEWT, así que dada la cadena TWENTYONE como entrada, tu algoritmo debería retornar 13.

(c) Cualquier cadena puede descomponerse en una secuencia de palíndromos. Por ejemplo, la cadena BUBBASEESABANANA ("Bubba ve una banana.") puede dividirse en palíndromos de las siguientes maneras (y 65 otras):

BUB • BASEESAB • ANANA

B • U • BB • ASEESA • B • ANANA

BUB • B • A • SEES • ABA • N • ANA

B • U • BB • A • S • EE • S • A • B • A • NAN • A

B • U • B • B • A • S • E • E • S • A • B • A • N • A • N • A

Describe y analiza un algoritmo eficiente para encontrar el menor número de palíndromos que conforman una cadena de entrada dada. Por ejemplo, dada la cadena de entrada BUBBASEESABANANA, tu algoritmo debería retornar 3.

(d) Describe y analiza un algoritmo eficiente para encontrar el entero más grande k tal que una cadena dada puede dividirse en palíndromos de longitud al menos k. Por ejemplo:

• Dada la cadena PALINDROME, tu algoritmo debería retornar 1. • Dada la cadena BUBBASEESABANANA, tu algoritmo debería retornar 3, para la partición BUB • BASEESAB • ANANA. • Dada una cadena de n símbolos idénticos, tu algoritmo debería retornar n.

(e) Describe y analiza un algoritmo eficiente para encontrar el número de diferentes formas en que una cadena dada puede descomponerse en palíndromos. Por ejemplo:

• Dada la cadena PALINDROME, tu algoritmo debería retornar 1. • Dada la cadena BUBBASEESABANANA, tu algoritmo debería retornar 70. • Dada una cadena de n símbolos idénticos, tu algoritmo debería retornar 2ⁿ⁻¹.

♥(f) Un **metapalíndromo** es una descomposición de una cadena en una secuencia de palíndromos, tal que la secuencia de longitudes de palíndromos es en sí misma un palíndromo. Por ejemplo:

BOB • S • MAM • ASEESA • UKU • L • ELE

es un metapalíndromo para la cadena BOBSMAMASEESAUKULELE, cuya secuencia de longitudes es el palíndromo (3, 1, 3, 6, 3, 1, 3). Describe y analiza un algoritmo eficiente para encontrar la longitud del metapalíndromo más corto para una cadena dada. Por ejemplo, dada la cadena de entrada BOBSMAMASEESAUKULELE, tu algoritmo debería retornar 11.

**10.** Supón que se te da un arreglo A[1 .. n] de enteros positivos. Una **subsecuencia creciente de ida y vuelta** es una secuencia de índices I[1 .. ℓ] con las siguientes propiedades:

• 1 ≤ I[j] ≤ n para todo j. • A[I[j]] < A[I[j + 1]] para todo j < ℓ. • Si I[j] es par, entonces I[j + 1] > I[j]. • Si I[j] es impar, entonces I[j + 1] < I[j].

Menos formalmente, supón que se nos da un arreglo de n cuadrados, cada uno conteniendo un entero positivo. Supón que colocamos un token en uno de los cuadrados, y luego movemos repetidamente el token a la izquierda (si está en un cuadrado de índice impar) o a la derecha (si está en un cuadrado de índice par), siempre moviéndose de un número menor a uno mayor. Entonces la secuencia de posiciones del token es una subsecuencia creciente de ida y vuelta.

Describe un algoritmo para calcular la longitud de la subsecuencia creciente de ida y vuelta más larga de un arreglo dado de n enteros. Por ejemplo, dado el arreglo de entrada

1 1 8 7 5 6 3 6 4 4 8 3 9 1 2 2 3 9 4 0

1< 2> 3< 4> 5< 6> 7< 8> 9< 10>11<12>13<14>15<16>17<18>19<20>

tu algoritmo debería retornar el entero 9, que es la longitud de la siguiente subsecuencia creciente de ida y vuelta:

0 1 2 3 4 6 7 8 9

20>1< 15<18>10>6> 4> 3<13<

**11.** Supón que queremos componer un párrafo de texto en una hoja de papel (o si insistes, una pantalla de computadora). El texto consiste en una secuencia de n palabras, donde la i-ésima palabra tiene longitud ℓ[i]. Queremos dividir el párrafo en varias líneas de longitud total exactamente L. Por ejemplo, según TEX, el programa usado para componer estas notas, el párrafo que estás leyendo ahora mide aproximadamente 11.94794 cm ≈ 4.7055 pulgadas de ancho.

Dependiendo de cómo se divida el párrafo en líneas de texto, debemos insertar diferentes cantidades de espacio en blanco entre las palabras. El párrafo debe estar completamente justificado, lo que significa que el primer carácter en cada línea comienza en el margen izquierdo, y excepto por la última línea, el último carácter en cada línea termina en el margen derecho. Debe haber al menos una unidad de espacio en blanco entre dos palabras cualesquiera en la misma línea. ¿Ves el párrafo que estás leyendo ahora? Exactamente así.

Define la **holgura** de un diseño de párrafo como la suma sobre todas las líneas, excepto la última, del cubo de la cantidad de espacio en blanco extra en cada línea, sin contar la unidad de espacio requerida entre cada par adyacente de palabras. Específicamente, si una línea contiene palabras i hasta j, entonces la holgura de esa línea se define como (L − j + i − ∑ₖ₌ᵢʲ ℓ[k])³. Describe un algoritmo de programación dinámica para imprimir el párrafo con holgura mínima.

**12.** Tú y tu sobrino Elmo de ocho años deciden jugar un juego de cartas simple. Al comienzo del juego, las cartas se reparten boca arriba en una fila larga. Cada carta vale un número diferente de puntos. Después de que todas las cartas se reparten, tú y Elmo se turnan para quitar ya sea la carta más a la izquierda o más a la derecha de la fila, hasta que todas las cartas se hayan ido. En cada turno, puedes decidir cuál de las dos cartas tomar. El ganador del juego es el jugador que ha recolectado más puntos cuando el juego termina.

Nunca habiendo tomado una clase de algoritmos, Elmo sigue la estrategia codiciosa obvia — cuando es su turno, Elmo siempre toma la carta con el valor de puntos más alto. Tu tarea es encontrar una estrategia que venza a Elmo cuando sea posible. (Podría parecer cruel vencer a un niño pequeño así, pero Elmo absolutamente odia cuando los adultos lo dejan ganar.)

(a) Demuestra que no deberías usar también la estrategia codiciosa. Es decir, muestra que hay un juego que puedes ganar, pero solo si no sigues la misma estrategia codiciosa que Elmo.

(b) Describe y analiza un algoritmo para determinar, dada la secuencia inicial de cartas, el número máximo de puntos que puedes recolectar jugando contra Elmo.

♣(c) Cuando Elmo tenía cuatro años, usaba una estrategia aún más simple — en su turno, siempre elegía su siguiente carta uniformemente al azar. Es decir, si había más de una carta restante en su turno, tomaría la carta más a la izquierda con probabilidad 1/2, y la carta más a la derecha con probabilidad 1/2. Describe un algoritmo para determinar, dada la secuencia inicial de cartas, el número máximo esperado de puntos que puedes recolectar jugando contra Elmo de cuatro años.

(d) Cinco años después, Elmo de trece años se ha convertido en un jugador mucho más fuerte. Describe y analiza un algoritmo para determinar, dada la secuencia inicial de cartas, el número máximo de puntos que puedes recolectar jugando contra un oponente perfecto.

**13.** ¡Es casi hora de mostrar tus increíbles habilidades de baile! Mañana es el gran concurso de baile para el que has estado entrenando toda tu vida, excepto por ese verano que pasaste con tu tío en Alaska cazando glotones. Has obtenido una copia anticipada de la lista de n canciones que los jueces tocarán durante el concurso, en orden cronológico. ¡Síííííííí!

Conoces todas las canciones, todos los jueces, y tu propia habilidad de baile extremadamente bien. Para cada entero k, sabes que si bailas la k-ésima canción en el programa, serás premiado con exactamente Score[k] puntos, pero entonces serás físicamente incapaz de bailar durante las siguientes Wait[k] canciones (es decir, no puedes bailar las canciones k + 1 hasta k + Wait[k]). El bailarín con la puntuación total más alta al final de la noche gana el concurso, así que quieres que tu puntuación total sea lo más alta posible.

Describe y analiza un algoritmo eficiente para calcular la puntuación total máxima que puedes lograr. La entrada a tu dulce algoritmo es el par de arreglos Score[1 .. n] y Wait[1 .. n].

**14.** El nuevo juego de rompecabezas de intercambio Candy Swap Saga XIII involucra n animales lindos numerados del 1 al n. Cada animal tiene uno de tres tipos de dulces: maníes de circo, barras Heath, y trufas de chocolate Cioccolateria Gardini. También tienes un dulce en tu mano; al inicio del juego, tienes un maní de circo.

Para ganar puntos, visitas cada uno de los animales en orden del 1 al n. Para cada animal, puedes ya sea mantener el dulce en tu mano o intercambiarlo con el dulce que el animal está sosteniendo.

• Si intercambias tu dulce por otro dulce del mismo tipo, ganas un punto. • Si intercambias tu dulce por un dulce de un tipo diferente, pierdes un punto. (Sí, tu puntuación puede ser negativa.) • Si visitas un animal y decides no intercambiar dulces, tu puntuación no cambia.

Debes visitar los animales en orden, y una vez que visitas un animal, nunca puedes visitarlo de nuevo.

Describe y analiza un algoritmo eficiente para calcular tu puntuación máxima posible. Tu entrada es un arreglo C[1 .. n], donde C[i] es el tipo de dulce que el i-ésimo animal está sosteniendo.

**15.** Lenny Rutenbar, el decano fundador del nuevo Maksymilian R. Levchin College of Computer Science, ha encargado una serie de rampas de nieve en la ladera sur de la colina de trineos Orchard Downs²⁰ y ha desafiado a Bill Kudeki, jefe del Departamento de Ingeniería Eléctrica y de Computación, a un concurso de trineo. Bill y Lenny van a deslizarse por la colina, cada uno tratando de maximizar su tiempo en el aire. El ganador puede expandir su departamento/colegio tanto al Siebel Center como al nuevo Edificio ECE; el perdedor tiene que mover todo su departamento/colegio al alcantarillado Boneyard junto al Laboratorio Loomis.

Cuando Lenny o Bill llega a una rampa mientras está en el suelo, pueden ya sea usar esa rampa para saltar por el aire, posiblemente volando sobre una o más rampas, o deslizarse más allá de esa rampa y permanecer en el suelo. Obviamente, si alguien vuela sobre una rampa, no puede usar esa rampa para extender su salto.

(a) Supón que se te da un par de arreglos Ramp[1 .. n] y Length[1 .. n], donde Ramp[i] es la distancia desde la cima de la colina hasta la i-ésima rampa, y Length[i] es la distancia que cualquier trineo que tome la i-ésima rampa viajará por el aire. Describe y analiza un algoritmo para determinar la distancia total máxima que Lenny o Bill pueden pasar en el aire.

(b) Los abogados de la universidad se enteraron de la pequeña apuesta de Lenny y Bill e inmediatamente objetaron. Para proteger a la universidad de demandas o tarifas de seguro que se disparen, imponen un límite superior en el número de saltos que cualquier trineo puede tomar. Describe y analiza un algoritmo para determinar la distancia total máxima que Lenny o Bill pueden pasar en el aire con a lo sumo k saltos, dados los arreglos originales Ramp[1 .. n] y Length[1 .. n] y el entero k como entrada.

♥(c) Cuando los abogados se dieron cuenta de que imponer su restricción no cerró inmediatamente el concurso, agregaron una nueva restricción: ¡Ninguna rampa puede usarse más de una vez! Disgustados por la interferencia legal, Lenny y Bill abandonan su apuesta y deciden cooperar para dar un buen espectáculo para los espectadores. Describe y analiza un algoritmo para determinar la distancia total máxima que Lenny y Bill pueden pasar en el aire, cada uno tomando a lo sumo k saltos (así que a lo sumo 2k saltos en total), y con cada rampa usada a lo sumo una vez.

**16.** Los granjeros Boggis, Bunce y Bean han establecido un curso de obstáculos para el Sr. Fox. El curso consiste en una fila larga de cabinas, cada una con un número pintado en el frente con pintura roja brillante. Formalmente, al Sr. Fox se le da un arreglo A[1 .. n], donde A[i] es el número pintado en el frente de la i-ésima cabina. Cada número A[i] podría ser positivo, negativo, o cero. Todos están de acuerdo con las siguientes reglas:

• En cada cabina, el Sr. Fox debe decir ya sea "¡Ring!" o "¡Ding!". • Si el Sr. Fox dice "¡Ring!" en la i-ésima cabina, gana una recompensa de A[i] pollos. (Si A[i] < 0, el Sr. Fox paga una penalidad de −A[i] pollos.) • Si el Sr. Fox dice "¡Ding!" en la i-ésima cabina, paga una penalidad de A[i] pollos. (Si A[i] < 0, el Sr. Fox gana una recompensa de −A[i] pollos.) • Al Sr. Fox se le prohíbe decir la misma palabra más de tres veces seguidas. Por ejemplo, si dice "¡Ring!" en las cabinas 6, 7 y 8, entonces debe decir "¡Ding!" en la cabina 9. • Todas las cuentas se liquidarán al final, después de que el Sr. Fox visite cada cabina y el árbitro grite "¡Hot box!" El Sr. Fox en realidad no tiene que cargar pollos por el curso de obstáculos. • Finalmente, si el Sr. Fox viola cualquiera de las reglas, o si termina el curso de obstáculos debiendo pollos a los granjeros, los granjeros le dispararán.

Describe y analiza un algoritmo para calcular el mayor número de pollos que el Sr. Fox puede ganar corriendo el curso de obstáculos, dado el arreglo A[1 .. n] de números como entrada. [Pista: ¡Cuidado con la piña ardiente!]

**17.** Dance Dance Revolution es un videojuego de baile, introducido por primera vez en Japón por Konami en 1998. Los jugadores se paran en una plataforma marcada con cuatro flechas, apuntando hacia adelante, atrás, izquierda y derecha, dispuestas en un patrón de cruz. Durante el juego, el juego toca una canción y desplaza una secuencia de n flechas (➜, ➜, ➜, o ➜) desde la parte inferior hasta la parte superior de la pantalla. En el momento preciso en que cada flecha llega a la parte superior de la pantalla, el jugador debe pisar la flecha correspondiente en la plataforma de baile. (Las flechas están cronometradas para que pises al ritmo de la canción.)

Estás jugando una variante de este juego llamada "Vogue Vogue Revolution", donde el objetivo es jugar perfectamente pero moverse lo menos posible. Cuando una flecha llega a la parte superior de la pantalla, si uno de tus pies ya está en la flecha correcta, se te otorga un punto de estilo por mantener tu pose actual. Si ningún pie está en la flecha correcta, debes mover un (y solo un) pie desde su ubicación actual hasta la flecha correcta en la plataforma. Si alguna vez pisas la flecha incorrecta, o fallas en pisar la flecha correcta, o mueves más de un pie a la vez, o mueves cualquier pie cuando ya estás parado en la flecha correcta, todos tus puntos de estilo son quitados y pierdes el juego.

¿Cómo deberías mover tus pies para maximizar tu número total de puntos de estilo? Para propósitos de este problema, asume que siempre comienzas con tu pie izquierdo en ➜ y tu pie derecho en ➜, y que has memorizado toda la secuencia de flechas. Por ejemplo, si la secuencia es ➜➜➜➜➜➜➜➜, puedes ganar 5 puntos de estilo moviendo tus pies como se muestra abajo:

**Referencia a diagrama de secuencia de movimientos de pies**

(a) Demuestra que para cualquier secuencia de n flechas, es posible ganar al menos n/4 − 1 puntos de estilo.

(b) Describe un algoritmo eficiente para encontrar el número máximo de puntos de estilo que puedes ganar durante una rutina VVR dada. La entrada a tu algoritmo es un arreglo Arrow[1 .. n] conteniendo la secuencia de flechas.

**18.** Considera la siguiente forma de solitario de Scrabble. Comenzamos con una secuencia fija y finita de fichas; cada ficha tiene tanto una letra como un valor numérico. Al inicio del juego, sacamos las primeras siete fichas de la secuencia y las ponemos en nuestra mano. En cada turno, formamos una palabra en inglés con algunas o todas las fichas en nuestra mano, colocamos esas fichas en la mesa, y recibimos el valor total de esas fichas como puntos. (Si no se puede formar ninguna palabra en inglés con las fichas en nuestra mano, el juego termina inmediatamente.) Entonces repetidamente sacamos la siguiente ficha del inicio de la secuencia hasta que ya sea (a) tenemos siete fichas en nuestra mano, o (b) la secuencia está vacía. (Lo siento, no hay puntuaciones dobles/triples de palabra/letra, bingos, comodines, o pasar.) Nuestro objetivo es obtener tantos puntos como sea posible.

Por ejemplo, considera la siguiente secuencia de 20 fichas:

I₂ N₂ X₈ A₁ N₂ A₁ D₃ U₅ D₃ I₂ D₃ K₈ U₅ B₄ L₂ A₁ K₈ H₅ A₁ N₂

Dada esta secuencia de fichas al comienzo del juego, podemos ganar 68 puntos de la siguiente manera:

• Inicialmente sacamos I₂ N₂ X₈ A₁ N₂ A₁ D₃. • Jugamos la palabra N₂ A₁ I₂ A₁ D₃ por 9 puntos, dejando N₂ X₈ en la mano. • Sacamos las siguientes cinco fichas U₅ D₃ I₂ D₃ K₈. • Jugamos la palabra U₅ N₂ D₃ I₂ D₃ por 15 puntos, dejando K₈ X₈ en la mano. • Sacamos las siguientes cinco fichas U₅ B₄ L₂ A₁ K₈. • Jugamos la palabra B₄ U₅ L₂ K₈ por 19 puntos, dejando K₈ X₈ A₁ en la mano. • Sacamos las últimas tres fichas H₅ A₁ N₂. • Jugamos la palabra A₁ N₂ K₈ H₅ por 16 puntos, dejando X₈ A₁ en la mano. • Jugamos la palabra A₁ X₈ por 9 puntos, vaciando nuestra mano y terminando el juego.

(a) Supón que la secuencia de fichas está representada por dos arreglos Letter[1 .. n], conteniendo una secuencia de letras entre A y Z, y Value[A .. Z], donde Value[ℓ] es el valor de cualquier ficha con letra ℓ. Diseña y analiza un algoritmo eficiente para calcular el número máximo de puntos que pueden ganarse de la secuencia dada de fichas.

(b) Ahora supón que dos fichas con la misma letra podrían tener valores diferentes. Ahora la secuencia de fichas está representada por dos arreglos Letter[1 .. n] y Value[1 .. n], donde Value[i] es el valor de la i-ésima ficha. Diseña y analiza un algoritmo eficiente para calcular el número máximo de puntos que pueden ganarse de la secuencia dada de fichas.

En ambos problemas, la salida es un solo número: la puntuación máxima posible. Asume (porque es verdad) que puedes encontrar todas las palabras en inglés que pueden hacerse con cualquier conjunto de a lo sumo siete fichas, junto con los valores de puntos de esas palabras, en tiempo O(1).

**19.** Supón que se nos da un conjunto L de n segmentos de línea en el plano, donde cada segmento tiene un extremo en la línea y = 0 y un extremo en la línea y = 1, y todos los 2n extremos son distintos.

(a) Describe y analiza un algoritmo para calcular el subconjunto más grande de L en el cual ningún par de segmentos se intersecta.

(b) Describe y analiza un algoritmo para calcular el subconjunto más grande de L en el cual cada par de segmentos se intersecta.

Ahora supón que se nos da un conjunto L de n segmentos de línea en el plano, donde ambos extremos de cada segmento yacen en el círculo unitario x² + y² = 1, y todos los 2n extremos son distintos.

(c) Describe y analiza un algoritmo para calcular el subconjunto más grande de L en el cual ningún par de segmentos se intersecta.

(d) Describe y analiza un algoritmo para calcular el subconjunto más grande de L en el cual cada par de segmentos se intersecta.

**20.** Sea P un conjunto de n puntos distribuidos uniformemente en el círculo unitario, y sea S un conjunto de m segmentos de línea con extremos en P. Los extremos de los m segmentos no son necesariamente distintos; n podría ser significativamente menor que 2m.

(a) Describe un algoritmo para encontrar el tamaño del subconjunto más grande de segmentos en S tal que cada par es disjunto. Dos segmentos son disjuntos si no se intersectan ni siquiera en sus extremos.

(b) Describe un algoritmo para encontrar el tamaño del subconjunto más grande de segmentos en S tal que cada par es interior-disjunto. Dos segmentos son interior-disjuntos si su intersección es ya sea vacía o un extremo de ambos segmentos.

(c) Describe un algoritmo para encontrar el tamaño del subconjunto más grande de segmentos en S tal que cada par se intersecta.

(d) Describe un algoritmo para encontrar el tamaño del subconjunto más grande de segmentos en S tal que cada par se cruza. Dos segmentos se cruzan si se intersectan pero no en sus extremos.

Para crédito completo, los cuatro algoritmos deberían funcionar en tiempo O(mn).

**21.** Estás manejando un autobús por una carretera, lleno de estudiantes ruidosos, hiperactivos y sedientos y una máquina de fuente de soda. Cada minuto que un estudiante está en tu autobús, ese estudiante bebe una onza de soda. Tu objetivo es dejar a los estudiantes rápidamente, para que la cantidad total de soda consumida por todos los estudiantes sea lo más pequeña posible.

Sabes cuántos estudiantes se bajarán del autobús en cada salida. Tu autobús comienza en algún lugar de la carretera (probablemente no en ningún extremo) y se mueve a una velocidad constante de 37.4 millas por hora. Debes manejar el autobús por la carretera; sin embargo, puedes manejar hacia adelante a una salida y luego hacia atrás a una salida en la dirección opuesta, cambiando tan a menudo como quieras. (Puedes parar el autobús, dejar estudiantes, y dar la vuelta instantáneamente.)

Describe un algoritmo eficiente para dejar a los estudiantes para que beban la menor soda posible. Tu entrada consiste en la ruta del autobús (una lista de las salidas, junto con el tiempo de viaje entre salidas sucesivas), el número de estudiantes que dejarás en cada salida, y la ubicación actual de tu autobús (que puedes asumir que es una salida).

**22.** Definamos un **resumen** de dos cadenas A y B como una concatenación de subcadenas de la siguiente forma:

• ÎSNA indica una subcadena SNA de solo la primera cadena A. • FOO indica una subcadena común FOO de ambas cadenas. • ÈBAR indica una subcadena BAR de solo la segunda cadena B.

Un resumen es válido si podemos recuperar las cadenas originales A y B concatenando las subcadenas apropiadas del resumen en orden y descartando los delimitadores Î, , y È. Cada carácter regular tiene longitud 1, y cada delimitador Î, , o È tiene alguna longitud no negativa fija Δ. La longitud de un resumen es la suma de las longitudes de sus símbolos.

Por ejemplo, cada una de las siguientes cadenas es un resumen válido de las cadenas KITTEN y KNITTING:

• KÈNITTÎEÈINÈG tiene longitud 9 + 7Δ. • KÈNITTÎENÈING tiene longitud 10 + 5Δ. • KÎITTENÈNITTING tiene longitud 13 + 3Δ. • ÎKITTENÈKNITTING tiene longitud 14 + 2Δ.

Describe y analiza un algoritmo que calcule la longitud del resumen más corto de dos cadenas dadas A[1 .. m] y B[1 .. n]. La longitud del delimitador Δ también es parte de la entrada a tu algoritmo. Por ejemplo:

• Dadas las cadenas KITTEN y KNITTING y Δ = 0, tu algoritmo debería retornar 9. • Dadas las cadenas KITTEN y KNITTING y Δ = 1, tu algoritmo debería retornar 15. • Dadas las cadenas KITTEN y KNITTING y Δ = 2, tu algoritmo debería retornar 18.

**23.** Vankin's Mile es un juego de solitario americano jugado en una cuadrícula cuadrada de n × n. El jugador comienza colocando un token en cualquier cuadrado de la cuadrícula. Entonces en cada turno, el jugador mueve el token ya sea un cuadrado a la derecha o un cuadrado hacia abajo. El juego termina cuando el jugador mueve el token fuera del borde del tablero. Cada cuadrado de la cuadrícula tiene un valor numérico, que podría ser positivo, negativo, o cero. El jugador comienza con una puntuación de cero; cuando el token aterriza en un cuadrado, el jugador agrega su valor a su puntuación. El objeto del juego es anotar tantos puntos como sea posible.

Por ejemplo, dada la cuadrícula de abajo, el jugador puede anotar 8−6+7−3+4 = 10 puntos colocando el token inicial en el 8 en la segunda fila, y luego moviéndose abajo, abajo, derecha, abajo, abajo. (Esta no es la mejor puntuación posible para esta cuadrícula de números.)

**Referencia a Figura con cuadrícula de juego**

(a) Describe y analiza un algoritmo eficiente para calcular la puntuación máxima posible para un juego de Vankin's Mile, dado el arreglo n× n de valores como entrada.

(b) En la versión europea de este juego, apropiadamente llamada Vankin's Kilometer, el jugador puede mover el token ya sea un cuadrado hacia abajo, un cuadrado a la derecha, o un cuadrado a la izquierda en cada turno. Sin embargo, para prevenir puntuaciones infinitas, el token no puede aterrizar en el mismo cuadrado más de una vez. Describe y analiza un algoritmo eficiente para calcular la puntuación máxima posible para un juego de Vankin's Kilometer, dado el arreglo n × n de valores como entrada.²¹

**Dividiendo Secuencias/Arreglos**

**32.** Una expresión aritmética básica está compuesta por caracteres del conjunto {1,+,×} y paréntesis. Casi todo entero puede representarse por más de una expresión aritmética básica. Por ejemplo, todas las siguientes expresiones aritméticas básicas representan el entero 14:

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

((1 + 1) × (1 + 1 + 1 + 1 + 1)) + ((1 + 1) × (1 + 1))

(1 + 1) × (1 + 1 + 1 + 1 + 1 + 1 + 1)

(1 + 1) × (((1 + 1 + 1) × (1 + 1)) + 1)

Describe y analiza un algoritmo para calcular, dado un entero n como entrada, el número mínimo de 1s en una expresión aritmética básica cuyo valor sea igual a n. El número de paréntesis no importa, solo el número de 1s. Por ejemplo, cuando n = 14, tu algoritmo debería retornar 8, para la expresión final de arriba. El tiempo de ejecución de tu algoritmo debería estar acotado por una función polinomial pequeña de n.

**33.** Supón que se te da una secuencia de enteros separados por signos + y −; por ejemplo:

1 + 3 − 2 − 5 + 1 − 6 + 7

Puedes cambiar el valor de esta expresión agregando paréntesis en diferentes lugares. Por ejemplo:

1 + 3 − 2 − 5 + 1 − 6 + 7 = −1

(1 + 3 − (2 − 5)) + (1 − 6) + 7 = 9

(1 + (3 − 2)) − (5 + 1) − (6 + 7) = −17

Describe y analiza un algoritmo para calcular, dada una lista de enteros separados por signos + y −, el valor máximo posible que la expresión puede tomar agregando paréntesis. Los paréntesis deben usarse solo para agrupar sumas y restas; en particular, no los uses para crear multiplicación implícita como en 1 + 3(−2)(−5) + 1 − 6 + 7 = 33.

**34.** Supón que se te da una secuencia de enteros separados por signos + y ×; por ejemplo:

1 + 3 × 2 × 0 + 1 × 6 + 7

Puedes cambiar el valor de esta expresión agregando paréntesis en diferentes lugares. Por ejemplo:

(1 + (3 × 2)) × 0 + (1 × 6) + 7 = 13

((1 + (3 × 2 × 0) + 1) × 6) + 7 = 19

(1 + 3) × 2 × (0 + 1) × (6 + 7) = 104

(a) Describe y analiza un algoritmo para calcular el valor máximo posible que la expresión dada puede tomar agregando paréntesis, asumiendo que todos los enteros en la entrada son positivos. [Pista: Esto es fácil.]

(b) Describe y analiza un algoritmo para calcular el valor máximo posible que la expresión dada puede tomar agregando paréntesis, asumiendo que todos los enteros en la entrada son no negativos.

(c) Describe y analiza un algoritmo para calcular el valor máximo posible que la expresión dada puede tomar agregando paréntesis, sin restricciones en los números de entrada.

Asume que cualquier operación aritmética toma tiempo O(1).

**35.** Después de graduarte de Sham-Poobanana University, decides entrevistarte para una posición en el banco de Wall Street Long Live Boole. El director gerente del banco, Eloob Egroeg, plantea un problema de 'resolver-o-morir' a cada nuevo empleado, que deben resolver en 24 horas. ¡Aquellos que fallan en resolver el problema son despedidos inmediatamente!

Al entrar al banco por primera vez, notas que las oficinas de empleados están organizadas en una fila recta, con una gran T o F impresa en la puerta de cada oficina. Además, entre cada par adyacente de oficinas, hay un tablero marcado por uno de los símbolos ∧, ∨, o ⊕. Cuando preguntas sobre estos símbolos arcanos, Eloob confirma que T y F representan los valores booleanos True y False, y los símbolos en los tableros representan los operadores booleanos estándar And, Or, y Xor. También explica que estas letras y símbolos describen si ciertas combinaciones de empleados pueden trabajar juntos exitosamente. Al inicio de cualquier nuevo proyecto, Eloob agrupa jerárquicamente a sus empleados agregando paréntesis a la secuencia de símbolos, para obtener una expresión booleana inequívoca. El proyecto es exitoso si esta expresión booleana con paréntesis evalúa a T.

Por ejemplo, si el banco tiene tres empleados, y la secuencia de símbolos en y entre sus puertas es T ∧ F ⊕ T, hay exactamente un esquema de parentización exitoso: (T ∧ (F ⊕ T)). Sin embargo, si la lista de símbolos de puertas es F ∧ T ⊕ F, no hay manera de agregar paréntesis para hacer el proyecto exitoso.

Eloob finalmente plantea tu pregunta de entrevista de resolver-o-morir: Describe un algoritmo para decidir si una secuencia dada de símbolos puede parentizarse para que la expresión booleana resultante evalúe a T. Tu entrada es un arreglo S[0 .. 2n], donde S[i] ∈ {T, F} cuando i es par, y S[i] ∈ {∨,∧,⊕} cuando i es impar.

**36.** Cada año, como parte de su reunión anual, los Amantes de Caracoles Antárticos de Upper Glacierville realizan una Carrera de Apareamiento de Mesa Redonda. Varios caracoles reproductores de alta calidad se colocan en el borde de una mesa redonda. Los caracoles están numerados en orden alrededor de la mesa del 1 al n. Durante la carrera, cada caracol vaga por la mesa, dejando un rastro de baba detrás. Los caracoles han sido especialmente entrenados para nunca caerse del borde de la mesa o cruzar un rastro de baba, ni siquiera el suyo propio. Si dos caracoles se encuentran, se declaran una pareja reproductora, se retiran de la mesa, y se los lleva a un agujero romántico en el suelo para hacer pequeños caracoles bebé. Nota que algunos caracoles pueden nunca encontrar pareja, incluso si la carrera continúa para siempre.

**Referencia a Figura 3.6**: El final de una carrera típica de SLUG Antártica. Los caracoles 6 y 8 nunca encuentran parejas. Los organizadores deben pagar M[3, 4] + M[2, 5] + M[1, 7].

Para cada par de caracoles, los organizadores de la carrera SLUG Antártica han publicado una recompensa monetaria, que se pagará a los dueños si ese par de caracoles se encuentra durante la Carrera de Apareamiento. Específicamente, hay un arreglo bidimensional M[1 .. n, 1 .. n] publicado en la pared detrás de la Mesa Redonda, donde M[i, j] = M[j, i] es la recompensa que se pagará si los caracoles i y j se encuentran.

Describe y analiza un algoritmo para calcular la recompensa total máxima que los organizadores podrían verse obligados a pagar, dado el arreglo M como entrada.

**37.** Has extraído una gran losa de mármol de una cantera. Para simplicidad, supón que la losa de mármol es un rectángulo que mide n pulgadas de altura y m pulgadas de ancho. Quieres cortar la losa en rectángulos más pequeños de varios tamaños — algunos para encimeras de cocina, algunos para proyectos de escultura grandes, otros para lápidas conmemorativas. Tienes una sierra de mármol que puede hacer cortes ya sea horizontales o verticales a través de cualquier losa rectangular. En cualquier momento, puedes consultar el precio al contado P[x, y] de un rectángulo de mármol de x pulgadas por y pulgadas, para cualquier enteros positivos x e y. Estos precios dependen de la demanda del cliente, y las personas que compran encimeras de mármol son raras, así que no hagas suposiciones sobre ellos; en particular, rectángulos más grandes pueden tener precios al contado significativamente menores. Dados el arreglo de precios al contado y los enteros m y n como entrada, describe un algoritmo para calcular cómo subdividir una losa de mármol n × m para maximizar tu ganancia.

**38.** Este problema te pide diseñar algoritmos eficientes para construir árboles de búsqueda binaria óptimos que satisfagan restricciones de balance adicionales. Tu entrada consiste en un arreglo ordenado A[1 .. n] de claves de búsqueda y un arreglo f[1 .. n] de conteos de frecuencia, donde f[i] es el número de búsquedas para A[i]. Esta es exactamente la misma función de costo descrita en la Sección 3.9. Pero ahora tu tarea es calcular un árbol óptimo que satisfaga algunas restricciones adicionales.

(a) Los árboles AVL fueron los primeros árboles de búsqueda binaria auto-balanceados, descritos por primera vez en 1962 por Georgy Adelson-Velsky y Evgenii Landis. Un árbol AVL es un árbol de búsqueda binaria donde para cada nodo v, la altura del subárbol izquierdo de v y la altura del subárbol derecho de v difieren por a lo sumo uno.

Describe y analiza un algoritmo para construir un árbol AVL óptimo para un conjunto dado de claves de búsqueda y frecuencias.

(b) Los árboles B binarios simétricos son otros árboles binarios auto-balanceados, descritos por primera vez por Rudolf Bayer en 1972; estos son mejor conocidos por el nombre árboles rojo-negro, después de una reformulación algo más simple por Leo Guibas y Bob Sedgwick en 1978. Un árbol rojo-negro es un árbol de búsqueda binaria con las siguientes restricciones adicionales:

• Cada nodo es ya sea rojo o negro. • Cada nodo rojo tiene un padre negro. • Cada camino de raíz a hoja contiene el mismo número de nodos negros.

Describe un algoritmo recursivo de retroceso para construir un árbol rojo-negro óptimo para un conjunto dado de claves de búsqueda y frecuencias.

(c) Los árboles AA fueron propuestos por Arne Andersson en 1993 y ligeramente simplificados (y nombrados) por Mark Allen Weiss en 2000. Los árboles AA también se conocen como árboles rojo-negro que se inclinan a la izquierda, después de una reformulación simétrica (con algoritmos de rebalanceo diferentes) por Bob Sedgewick en 2006. Un árbol AA es un árbol rojo-negro con una restricción adicional:

• Ningún hijo izquierdo es rojo.²³

Describe y analiza un algoritmo para construir un árbol AA óptimo para un conjunto dado de claves de búsqueda y frecuencias.

**39.** Supón que se te da un mapa de bits m × n como un arreglo M[1 .. m, 1 .. n] de 0s y 1s. Un bloque sólido en M es un subarreglo de la forma M[i .. i′, j .. j′] en el cual todos los bits son iguales. Supón que quieres descomponer M en la menor cantidad posible de bloques disjuntos.

Una estrategia natural de partición recursiva se llama **subdivisión guillotina**. Si todo el mapa de bits M es un bloque sólido, no hay nada que hacer. De otro modo, cortamos M en dos mapas de bits más pequeños a lo largo de una línea horizontal o vertical, y luego descomponemos recursivamente los dos mapas de bits más pequeños en bloques sólidos.

Cualquier subdivisión guillotina puede representarse como un árbol binario, donde cada nodo interno almacena la posición y orientación de un corte, y cada hoja almacena un solo bit 0 o 1 indicando el contenido del bloque correspondiente. El tamaño de una subdivisión guillotina es el número de hojas en el árbol binario correspondiente (es decir, el número final de bloques sólidos), y la profundidad de una subdivisión guillotina es la profundidad del árbol binario correspondiente.

(a) Describe y analiza un algoritmo para calcular una subdivisión guillotina de M del tamaño mínimo posible.

(b) Muestra que una subdivisión guillotina no siempre produce una partición en el menor número de bloques sólidos.

(c) Describe y analiza un algoritmo para calcular una subdivisión guillotina para M con la menor profundidad posible.

(d) Describe y analiza un algoritmo para determinar M[i, j], dado el árbol que representa una descomposición guillotina para M y dos índices i y j.

**Referencia a Figura 3.7**: Una subdivisión guillotina con tamaño 8 y profundidad 5.

(e) Define la profundidad de un píxel M[i, j] en una subdivisión guillotina como la profundidad de la hoja que contiene ese píxel. Describe y analiza un algoritmo para calcular una subdivisión guillotina para M tal que la suma de las profundidades de los píxeles sea lo más pequeña posible.

(f) Describe y analiza un algoritmo para calcular una subdivisión guillotina para M tal que la suma de las profundidades de los píxeles negros sea lo más pequeña posible.

♠**40.** ¡Felicitaciones! Has sido contratado por la librería en línea gigante DeNile ("¡No solo un río en Egipto!") para optimizar sus robots de almacén. Cada libro que DeNile vende tiene un ISBN único (Número Estándar Internacional de Libro), que es solo un valor numérico. Cada uno de los almacenes de DeNile contiene una fila larga de contenedores, cada uno conteniendo múltiples copias de un solo libro. Estos contenedores están organizados en orden ordenado por ISBN; el ISBN de cada contenedor está impreso en el frente del contenedor en forma legible por máquina. Los libros se recuperan de estos contenedores por robots, que corren a lo largo de rieles paralelos a la fila de contenedores.

DeNile no mantiene una lista de qué contenedores contienen qué números ISBN; ¡eso sería demasiado simple! En su lugar, para recuperar un libro deseado, el robot debe primero encontrar el contenedor de ese libro usando una búsqueda binaria. Porque la búsqueda requiere movimiento físico por parte del robot, ya no podemos asumir que cada paso de la búsqueda binaria requiere tiempo O(1). Específicamente:

• El robot siempre comienza en el "contenedor 0" (donde los libros se cargan en cajas para enviar a los clientes). • Mover el robot del i-ésimo contenedor al j-ésimo contenedor requiere α|i − j| segundos para alguna constante α. • El robot debe estar directamente frente a un contenedor para leer el ISBN de ese contenedor. Leer un ISBN requiere β segundos, para alguna constante β. • Revertir la dirección de movimiento del robot (de creciente a decreciente o viceversa) requiere γ segundos adicionales, para alguna constante γ. • Cuando el robot encuentra el contenedor objetivo, extrae un libro de ese contenedor y regresa al "contenedor 0".

Diseña y analiza un algoritmo para calcular un árbol de búsqueda binaria sobre los contenedores que minimice el tiempo total que el robot gasta buscando libros. Tu entrada es un arreglo f[1 .. n] de enteros, donde f[i] es el número de veces que se le pedirá al robot recuperar un libro del i-ésimo contenedor, junto con los parámetros de tiempo α, β, y γ.

♠**41.** Un método estándar para mejorar el rendimiento de caché de árboles de búsqueda es empacar más claves de búsqueda y subárboles en cada nodo. Un B-árbol es un árbol con raíz en el cual cada nodo interno almacena hasta B claves y punteros a hasta B + 1 hijos, cada uno la raíz de un B-árbol más pequeño. Específicamente, cada nodo v almacena tres campos:

• un entero positivo v.d ≤ B, • un arreglo ordenado v.key[1 .. v.d], y • un arreglo v.child[0 .. v.d] de punteros hijo.

En particular, el número de punteros hijo es siempre exactamente uno más que el número de claves.²⁴

Cada puntero v.child[i] es ya sea Null o un puntero a la raíz de un B-árbol cuyas claves son todas mayores que v.key[i] y menores que v.key[i + 1]. En particular, todas las claves en el subárbol más a la izquierda v.child[0] son menores que v.key[1], y todas las claves en el subárbol más a la derecha v.child[v.d] son mayores que v.key[v.d].

Intuitivamente, deberías tener la siguiente imagen en mente:

[ •·⚬ < key[1] < •·⚬ < key[2] < •·⚬ ··· •·⚬ < key[d] < •·⚬ ]

T₀ T₁ T₂ ··· Tₐ₋₁ Tₐ

Aquí Tᵢ es el subárbol apuntado por child[i].

El costo de buscar una clave x en un B-árbol es el número de nodos en el camino desde la raíz hasta el nodo que contiene x como una de sus claves. Un 1-árbol es solo un árbol de búsqueda binaria estándar.

Fija un entero positivo arbitrario B > 0. (Sugiero B = 8.) Supón que se te da un arreglo ordenado A[1, . . . , n] de claves de búsqueda y un arreglo correspondiente F[1, . . . , n] de conteos de frecuencia, donde F[i] es el número de veces que buscaremos A[i]. Tu tarea es describir y analizar un algoritmo eficiente para encontrar un B-árbol que minimice el costo total de buscar las claves dadas con las frecuencias dadas.

(a) Describe un algoritmo de tiempo polinomial para el caso especial B = 2.

(b) Describe un algoritmo para B arbitrario que funciona en tiempo O(n^(B+c)) para algún entero fijo c.

♥(c) Describe un algoritmo para B arbitrario que funciona en tiempo O(n^c) para algún entero fijo c que no depende de B.

**42.** Una cadena w de paréntesis ( y ) y corchetes [ y ] está **balanceada** si satisface una de las siguientes condiciones:

• w es la cadena vacía. • w = (x) para alguna cadena balanceada x • w = [x] para alguna cadena balanceada x • w = x y para algunas cadenas balanceadas x e y

Por ejemplo, la cadena

w = ([()][]()) [()()] ()

está balanceada, porque w = x y, donde

x = ( [()] [] () ) y y = [ () () ] ().

(a) Describe y analiza un algoritmo para determinar si una cadena dada de paréntesis y corchetes está balanceada.

(b) Describe y analiza un algoritmo para calcular la longitud de una subsecuencia balanceada más larga de una cadena dada de paréntesis y corchetes.

(c) Describe y analiza un algoritmo para calcular la longitud de una supersecuencia balanceada más corta de una cadena dada de paréntesis y corchetes.

(d) Describe y analiza un algoritmo para calcular la distancia de edición mínima de una cadena dada de paréntesis y corchetes a una cadena balanceada de paréntesis y corchetes.

♥(e) Describe y analiza un algoritmo para calcular la subsecuencia común balanceada más larga de dos cadenas dadas de paréntesis y corchetes.

♥(f) Describe y analiza un algoritmo para calcular la subsecuencia balanceada palindrómica más larga de una cadena dada de paréntesis y corchetes.

♥(g) Describe y analiza un algoritmo para calcular la subsecuencia común palindrómica balanceada más larga (¡uf!) de dos cadenas dadas de paréntesis y corchetes.

Para cada problema, tu entrada es un arreglo w[1 .. n], donde w[i] ∈ {(, ), [, ]} para cada índice i. (Puedes preferir usar símbolos diferentes en lugar de paréntesis y corchetes — por ejemplo, L, R, l, r o Ã,Â,Ê,É — ¡pero por favor dile a tu calificador qué símbolos estás usando!)

♥**43.** ¡Felicitaciones! Tu equipo de investigación acaba de recibir un proyecto multianual de $50M, financiado conjuntamente por DARPA, Google, y McDonald's, para producir DWIM: ¡El primer compilador que lee la mente de los programadores! Tu propuesta y tus numerosos comunicados de prensa prometen que DWIM corregirá automáticamente errores en cualquier pieza de código dada, mientras modifica ese código lo menos posible. Desafortunadamente, ahora es hora de comenzar a hacer que la maldita cosa funcione.

Como ejercicio de calentamiento, decides abordar el siguiente subproblema necesario. Recuerda que la distancia de edición entre dos cadenas es el número mínimo de inserciones, eliminaciones y reemplazos de caracteres individuales requerido para transformar una cadena en la otra. Una expresión aritmética es una cadena w tal que

• w es una cadena de uno o más dígitos decimales, • w = (x) para alguna expresión aritmética x, o • w = x ⊙ y para algunas expresiones aritméticas x e y y algún operador binario ⊙.

Supón que se te da una cadena de tokens del alfabeto {#, ⊙, (, )}, donde # representa un dígito decimal y ⊙ representa un operador binario. Describe y analiza un algoritmo para calcular la distancia de edición mínima de la cadena dada a una expresión aritmética.

**44.** Las moléculas de ácido ribonucleico (ARN) son cadenas largas de millones de nucleótidos o bases de cuatro tipos diferentes: adenina (A), citosina (C), guanina (G), y uracilo (U). La secuencia de una molécula de ARN es una cadena b[1 .. n], donde cada carácter b[i] ∈ {A, C, G, U} corresponde a una base. Además de los enlaces químicos entre bases adyacentes en la secuencia, los enlaces de hidrógeno pueden formarse entre ciertos pares de bases. El conjunto de pares de bases enlazados se llama la estructura secundaria de la molécula de ARN.

Decimos que dos pares de bases (i, j) y (i′, j′) con i < j e i′ < j′ se superponen si i < i′ < j < j′ o i′ < i < j′ < j. En la práctica, la mayoría de los pares de bases no se superponen. Los pares de bases superpuestos crean los llamados pseudonudos en la estructura secundaria, que son esenciales para algunas funciones del ARN, pero son más difíciles de predecir.

Supón que queremos predecir la mejor estructura secundaria posible para una secuencia de ARN dada. Adoptaremos un modelo drásticamente simplificado de estructura secundaria:

• Cada base puede enlazarse con a lo sumo una otra base. • Solo los pares A–U y C–G pueden enlazarse. • Los pares de la forma (i, i + 1) y (i, i + 2) no pueden enlazarse. • Los pares de bases enlazados no pueden superponerse.

La última (y menos realista) restricción nos permite visualizar la estructura secundaria del ARN como una especie de árbol gordo, como se muestra abajo.

**Referencia a Figura 3.8**: Ejemplo de estructura secundaria de ARN con 21 pares de bases enlazados, indicados por líneas rojas gruesas. Los espacios se indican por curvas punteadas. Esta estructura tiene puntuación 2² + 2² + 8² + 1² + 7² + 4² + 7² = 187.

(a) Describe y analiza un algoritmo que calcule el número máximo posible de pares de bases enlazados en una estructura secundaria para una secuencia de ARN dada.

(b) Un espacio en una estructura secundaria es una subcadena maximal de bases no apareadas. Los espacios grandes llevan a inestabilidades químicas, así que las estructuras secundarias con espacios menores son más probables. Para dar cuenta de esta preferencia, definamos la puntuación de una estructura secundaria como la suma de los cuadrados de las longitudes de los espacios; ver Figura 3.8. (Esta función de puntuación es completamente ficticia; la predicción real de estructura de ARN requiere funciones de puntuación mucho más complicadas.)

Describe y analiza un algoritmo que calcule la puntuación mínima posible de una estructura secundaria para una secuencia de ARN dada.

♣**45.** (a) Describe y analiza un algoritmo eficiente para determinar, dada una cadena w y una expresión regular R, si w ∈ L(R).

(b) Las expresiones regulares generalizadas permiten el operador binario ∩ (intersección) y el operador unario ¬ (complemento), además de los operadores usuales • (concatenación), + (o), y \* (cerradura de Kleene). Las construcciones NFA y el teorema de Kleene implican que cualquier expresión regular generalizada E representa un lenguaje regular L(E).

Describe y analiza un algoritmo eficiente para determinar, dada una cadena w y una expresión regular generalizada E, si w ∈ L(E).

En ambos problemas, asume que en realidad se te da un árbol de análisis para la expresión regular (generalizada), no solo una cadena.

**Árboles y Subárboles**

**46.** Acabas de ser nombrado como el nuevo organizador de la primera fiesta navideña anual obligatoria en Giggle (una subsidiaria de Abugida). Los empleados de Giggle están organizados en una jerarquía estricta — un árbol con el presidente de la compañía en la raíz. Los oráculos omniscientes en Recursos Humanos han asignado un número real a cada empleado midiendo qué tan "divertido" es el empleado. Para mantener las cosas sociales, hay una restricción en la lista de invitados: un empleado no puede asistir a la fiesta si su supervisor inmediato también está presente. Por otro lado, el presidente de la compañía debe asistir a la fiesta, aunque tenga una calificación de diversión negativa; es su compañía, después de todo. Da un algoritmo que haga una lista de invitados para la fiesta que maximice la suma de las calificaciones de "diversión" de los invitados.

**47.** Como tan pocas personas vinieron a la fiesta navideña del año pasado, el presidente de Giggle decide dar a cada empleado un regalo en su lugar este año. Específicamente, cada empleado debe recibir uno de los tres regalos: (1) unas vacaciones de seis semanas con todos los gastos pagados en cualquier lugar del mundo, (2) un desayuno de todos los panqueques que puedas ordenar para dos en Jumping Jack Flash's Flapjack Stack Shack, o (3) una bolsa de papel ardiendo llena de excremento de perro. Las regulaciones corporativas prohíben que cualquier empleado reciba exactamente el mismo regalo que su supervisor directo. Cualquier empleado que reciba un regalo mejor que su supervisor directo casi ciertamente será despedido en un ataque de celos.

Como zar oficial de la fiesta de Giggle, es tu trabajo decidir qué regalo recibe cada empleado. Describe un algoritmo para distribuir regalos para que el número mínimo de personas sean despedidas. Sí, puedes enviarle al presidente una bolsa ardiente de excremento de perro.

Más formalmente, se te da un árbol con raíz T, representando la jerarquía de la compañía, y quieres etiquetar los nodos de T con enteros 1, 2, o 3, para que cada nodo tenga una etiqueta diferente de su padre. El costo de un etiquetado es el número de nodos con etiquetas menores que sus padres. Ver Figura 3.9 para un ejemplo. Describe y analiza un algoritmo para calcular el etiquetado de costo mínimo de T.

**Referencia a Figura 3.9**: Un etiquetado de árbol con costo 9. Los nueve nodos en negrita tienen etiquetas menores que sus padres. Este no es el etiquetado óptimo para este árbol.

**48.** Después del Debacle Navideño del Excremento de Perro Ardiente, fuiste fuertemente alentado a buscar otro empleo, así que dejaste Giggle por la compañía rival Twitbook. Desafortunadamente, el nuevo presidente de Twitbook acaba de decidir imitar a Giggle organizando su propia fiesta navideña, y en vista de tu experiencia pasada, te nombró como el organizador oficial de la fiesta. El presidente exige que invites exactamente k empleados, incluyendo al presidente mismo, y todos los que son invitados están obligados a asistir. Sí, eso será divertido.

Igual que en Giggle, los empleados en Twitbook están organizados en una jerarquía estricta: un árbol con el presidente de la compañía en la raíz. Los oráculos omniscientes en Recursos Humanos han asignado un número real a cada empleado indicando la incomodidad de invitar tanto a ese empleado como a su supervisor inmediato; un valor negativo indica que el empleado y su supervisor en realidad se gustan. Tu objetivo es elegir un subconjunto de exactamente k empleados para invitar, para que la incomodidad total de la fiesta resultante sea lo más pequeña posible. Por ejemplo, si la lista de invitados no incluye tanto a un empleado como a su supervisor inmediato, la incomodidad total es cero. La entrada a tu algoritmo es el árbol T, el entero k, y la incomodidad de cada nodo en T.

(a) Describe un algoritmo que calcule la incomodidad total del subconjunto menos incómodo de k empleados, asumiendo que la jerarquía de la compañía está descrita por un árbol binario. Es decir, asume que cada empleado supervisa directamente a lo sumo dos otros.

♥(b) Describe un algoritmo que calcule la incomodidad total del subconjunto menos incómodo de k empleados, sin restricciones en la jerarquía de la compañía.

**49.** Supón que necesitamos transmitir un mensaje a todos los nodos en un árbol con raíz. Inicialmente, solo el nodo raíz conoce el mensaje. En una sola ronda, cualquier nodo que conoce el mensaje puede reenviarlo a lo sumo uno de sus hijos. Ver Figura 3.10 para un ejemplo.

(a) Diseña un algoritmo para calcular el número mínimo de rondas requeridas para transmitir el mensaje a todos los nodos en un árbol binario.

♦(b) Diseña un algoritmo para calcular el número mínimo de rondas requeridas para transmitir el mensaje a todos los nodos en un árbol con raíz arbitrario. [Pista: Puedes encontrar útiles las técnicas en el siguiente capítulo para probar que tu algoritmo es correcto, aunque no es un algoritmo codicioso.]

**Referencia a Figura 3.10**: Un mensaje siendo distribuido a través de un árbol en cinco rondas.

**50.** Un día, Alex se cansó de escalar en un gimnasio y decidió llevar a un grupo muy grande de amigos escaladores afuera a escalar. El área de escalada a donde fueron tenía una roca enorme y ancha, no muy alta, con varias presas de mano y pie marcadas. Alex rápidamente determinó un conjunto "permitido" de movimientos que su grupo de amigos puede realizar para ir de una presa a otra.

El sistema general de presas puede describirse por un árbol con raíz T con n vértices, donde cada vértice corresponde a una presa y cada arista corresponde a un movimiento permitido entre presas. Los caminos de escalada convergen cuando suben por la roca, llevando a una presa única en la cima, representada por la raíz de T.²⁵

Alex y sus amigos (que son todos escaladores excelentes) decidieron jugar un juego, donde tantos escaladores como sea posible están simultáneamente en la roca y cada escalador necesita realizar una secuencia de exactamente k movimientos. Cada escalador puede elegir una presa arbitraria para comenzar, y todos los movimientos deben alejarse del suelo. Así, cada escalador traza un camino de k aristas en el árbol T, todas dirigidas hacia la raíz. Sin embargo, no se permite que dos escaladores toquen la misma presa; los caminos seguidos por diferentes escaladores no pueden intersectar en absoluto.

Describe y analiza un algoritmo eficiente para calcular el número máximo de escaladores que pueden jugar este juego. Más formalmente, se te da un árbol con raíz T y un entero k, y quieres encontrar el mayor número posible de caminos disjuntos en T, donde cada camino tiene longitud k. No asumas que T es un árbol binario. Por ejemplo, dado el árbol T de abajo y k = 3 como entrada, tu algoritmo debería retornar el entero 8.

**Referencia a Figura 3.11**: Siete caminos disjuntos de longitud k = 3. Este no es el conjunto más grande de tales caminos en este árbol.

**51.** Sea T un árbol binario con raíz con n vértices, y sea k ≤ n un entero positivo. Nos gustaría marcar k vértices en T para que cada vértice tenga un ancestro marcado cercano. Más formalmente, definimos el costo de agrupamiento de cualquier subconjunto K de vértices como

cost(K) = max\_v cost(v, K),

donde el máximo se toma sobre todos los vértices v en el árbol, y cost(v, K) es la distancia de v a su ancestro más cercano en K:

cost(v, K) = {

0 si v ∈ K

∞ si v es la raíz de T y v ∉ K

1 + cost(parent(v)) en caso contrario

}

En particular, cost(K) = ∞ si K excluye la raíz de T.

**Referencia a Figura 3.12**: Un subconjunto de cinco vértices en un árbol binario, con costo de agrupamiento 3.

♥(a) Describe un algoritmo de programación dinámica para calcular, dado el árbol T y un entero k, el costo mínimo de agrupamiento de cualquier subconjunto de k vértices en T. Para crédito completo, tu algoritmo debería funcionar en tiempo O(n²k²).

(b) Describe un algoritmo de programación dinámica para calcular, dado el árbol T y un entero r, el tamaño del subconjunto más pequeño de vértices cuyo costo de agrupamiento es a lo sumo r. Para crédito completo, tu algoritmo debería funcionar en tiempo O(nr).

(c) Muestra que tu solución para la parte (b) implica un algoritmo para la parte (a) que funciona en tiempo O(n² log n).

**52.** Esta pregunta te pide encontrar algoritmos eficientes para calcular el subárbol con raíz común más grande de dos árboles con raíz dados. Recuerda que un árbol con raíz es un grafo acíclico conectado con un nodo designado llamado la raíz. Un subárbol con raíz de un árbol con raíz consiste en un nodo arbitrario y todos sus descendientes. La definición precisa de "común" depende de qué pares de árboles con raíz consideramos isomorfos.

(a) Recuerda que un árbol binario es un árbol con raíz en el cual cada nodo tiene un subárbol izquierdo (posiblemente vacío) y un subárbol derecho (posiblemente vacío). Dos árboles binarios son isomorfos si y solo si ambos están vacíos, o sus subárboles izquierdos son isomorfos y sus subárboles derechos son isomorfos.

Describe un algoritmo para encontrar el subárbol binario común más grande de dos árboles binarios dados.

**Referencia a Figura 3.13**: Dos árboles binarios, con su subárbol común (con raíz) más grande enfatizado.

(b) En un árbol con raíz ordenado, cada nodo tiene una secuencia de hijos, que son las raíces de subárboles con raíz ordenados. Dos árboles con raíz ordenados son isomorfos si ambos están vacíos, o si sus i-ésimos subárboles son isomorfos para cada índice i. Describe un algoritmo para encontrar el subárbol ordenado común más grande de dos árboles ordenados T₁ y T₂.

♦♥(c) En un árbol con raíz no ordenado, cada nodo tiene un conjunto no ordenado de hijos, que son las raíces de subárboles con raíz no ordenados. Dos árboles con raíz no ordenados son isomorfos si ambos están vacíos, o los subárboles de cada raíz pueden ordenarse para que sus i-ésimos subárboles sean isomorfos para cada índice i. Describe un algoritmo para encontrar el subárbol no ordenado común más grande de dos árboles no ordenados T₁ y T₂.

**53.** Esta pregunta te pide encontrar algoritmos eficientes para calcular subárboles óptimos en árboles sin raíz — grafos no dirigidos acíclicos conectados. Un subárbol de un árbol sin raíz es cualquier subgrafo conectado.

(a) Supón que se te da un árbol sin raíz T con pesos en sus aristas, que pueden ser positivos, negativos, o cero. Describe un algoritmo para encontrar un camino en T con peso total máximo.

(b) Supón que se te da un árbol sin raíz T con pesos en sus vértices, que pueden ser positivos, negativos, o cero. Describe un algoritmo para encontrar un subárbol de T con peso total máximo. [Esta fue una pregunta de entrevista de Google 2016.]

(c) Sean T₁ y T₂ árboles sin raíz ordenados arbitrarios, lo que significa que los vecinos de cada nodo tienen un orden cíclico bien definido. Describe un algoritmo para encontrar el subárbol ordenado común más grande de T₁ y T₂.

♦♥(d) Sean T₁ y T₂ árboles sin raíz no ordenados arbitrarios. Describe un algoritmo para encontrar el subárbol no ordenado común más grande de T₁ y T₂.

**54.** Los menores con raíz de árboles con raíz son una generalización natural de las subsecuencias. Un menor con raíz de un árbol con raíz T es cualquier árbol obtenido contrayendo una o más aristas. Cuando contraemos una arista uv, donde u es el padre de v, los hijos de v se convierten en nuevos hijos de u y luego v se elimina. En particular, la raíz de T es también la raíz de cada menor con raíz de T.

**Referencia a Figura 3.14**: Un árbol con raíz y uno de sus menores con raíz.

(a) Sea T un árbol con raíz con nodos etiquetados. Decimos que T es **aburrido** si, para cada nodo x, todos los hijos de x tienen la misma etiqueta; los hijos de nodos diferentes pueden tener etiquetas diferentes. Describe un algoritmo para encontrar el menor con raíz aburrido más grande de un árbol con raíz etiquetado dado.

(b) Supón que se nos da un árbol con raíz T cuyos nodos están etiquetados con números. Describe un algoritmo para encontrar el menor con raíz ordenado como heap más grande de T. Es decir, tu algoritmo debería retornar el menor con raíz más grande M tal que cada nodo en M tenga una etiqueta menor que sus hijos en M.

**Notas al pie**

¹⁸ En mis clases de algoritmos, cualquier solución de programación dinámica que no incluya una especificación en inglés de los subproblemas recursivos subyacentes automáticamente obtiene una puntuación de cero, incluso si la solución es por lo demás perfecta. Introducir esta política mejoró significativamente las calificaciones de los estudiantes, porque redujo significativamente el número de veces que enviaron algoritmos de programación dinámica incorrectos (o incoherentes).

¹⁹ Para más detalles sobre la historia y cultura de Nadiria, incluyendo imágenes de las varias denominaciones de Dream-Dollars, ver http://moneyart.biz/dd/.

²⁰ La ladera norte es más rápida, pero demasiado corta para un concurso interesante.

²¹ Si también permitiéramos movimiento hacia arriba, el juego resultante (¿Vankin's Fathom?) sería NP-difícil.

**24.** Supón que se te da un mapa de bits m × n como un arreglo M[1 .. n, 1 .. n] de 0s y 1s. Un bloque sólido en M es un subarreglo de la forma M[i .. i′, j .. j′] en el cual todos los bits son iguales. Un bloque sólido es cuadrado si tiene el mismo número de filas y columnas.

(a) Describe un algoritmo para encontrar el área máxima de un bloque cuadrado sólido en M en tiempo O(n²).

(b) Describe un algoritmo para encontrar el área máxima de un bloque sólido en M en tiempo O(n³).

(c) Describe un algoritmo para encontrar el área máxima de un bloque sólido en M en tiempo O(n² log n). [Pista: Divide y vencerás.]

♥(d) Describe un algoritmo para encontrar el área máxima de un bloque sólido en M en tiempo O(n²).

**25.** Supón que se te da un arreglo M[1 .. n, 1 .. n] de números, que pueden ser positivos, negativos, o cero, y que no son necesariamente enteros. Describe un algoritmo para encontrar la suma más grande de elementos en cualquier subarreglo rectangular de la forma M[i .. i′, j .. j′]. Para crédito completo, tu algoritmo debería funcionar en tiempo O(n³). [Pista: Ver problema 3.]

**26.** Describe y analiza un algoritmo que encuentre el patrón rectangular de área máxima que aparece más de una vez en un mapa de bits dado. Específicamente, dado un arreglo bidimensional M[1 .. n, 1 .. n] de bits como entrada, tu algoritmo debería producir el área del patrón rectangular repetido más grande en M. Por ejemplo, dado el mapa de bits mostrado a la izquierda en la figura de abajo, tu algoritmo debería retornar el entero 195, que es el área del perrito 15 × 13. (Aunque no pasa en este ejemplo, las dos copias del patrón repetido podrían superponerse.)

(a) Para crédito completo, describe un algoritmo que funcione en tiempo O(n⁵).

♥(b) Para crédito extra, describe un algoritmo que funcione en tiempo O(n⁴).

♣♥(c) Para crédito extra extra, describe un algoritmo que funcione en tiempo O(n³ polylog n).

**27.** Sea P un conjunto de puntos en el plano en posición convexa. Intuitivamente, si una banda elástica fuera envuelta alrededor de los puntos, entonces cada punto tocaría la banda elástica. Más formalmente, para cualquier punto p en P, hay una línea que separa p de los otros puntos en P. Además, supón que los puntos están indexados P[1], P[2], . . . , P[n] en orden antihorario alrededor de la "banda elástica", comenzando con el punto más a la izquierda P[1].

Este problema te pide resolver un caso especial del problema del vendedor viajero, donde el vendedor debe visitar cada punto en P, y el costo de moverse de un punto p ∈ P a otro punto q ∈ P es la distancia euclidiana |pq|.

(a) Describe un algoritmo simple para calcular el tour cíclico más corto de P.

(b) Un tour simple es uno que nunca se cruza a sí mismo. Demuestra que el tour más corto de P debe ser simple.

(c) Describe y analiza un algoritmo eficiente para calcular el tour más corto de P que comienza en el punto más a la izquierda P[1] y termina en el punto más a la derecha P[r].

(d) Describe y analiza un algoritmo eficiente para calcular el tour más corto de P, sin restricciones en los puntos finales.

♥**28.** Describe y analiza un algoritmo para resolver el problema del vendedor viajero en tiempo O(2ⁿ poly(n)). Dado un grafo no dirigido de n vértices G con aristas pesadas, tu algoritmo debería retornar el peso del ciclo más ligero en G que visita cada vértice exactamente una vez, o ∞ si G no tiene tales ciclos. [Pista: El algoritmo recursivo de retroceso obvio toma tiempo O(n!).]

**29.** Sea W = {w₁, w₂, . . . , wₙ} un conjunto finito de cadenas sobre algún alfabeto fijo Σ. Un **centro de edición** para W es una cadena C ∈ Σ\* tal que la distancia de edición máxima de C a cualquier cadena en W sea lo más pequeña posible. El **radio de edición** de W es la distancia de edición máxima de un centro de edición a una cadena en W. Un conjunto de cadenas puede tener varios centros de edición, pero su radio de edición es único.

EditRadius(W) := min\_{C∈Σ\*} max\_{w∈W} Edit(w, C)

EditCenter(W) := argmin\_{C∈Σ\*} max\_{w∈W} Edit(w, C)

(a) Describe y analiza un algoritmo eficiente para calcular el radio de edición de tres cadenas dadas.

♣♥(b) Describe y analiza un algoritmo eficiente para aproximar el radio de edición de un conjunto arbitrario de cadenas dentro de un factor de 2. (Calcular el radio de edición exacto es NP-difícil a menos que el número de cadenas sea fijo.)

♥**30.** Sea D[1 .. n] un arreglo de dígitos, cada uno un entero entre 0 y 9. Una **subsecuencia digital** de D es una secuencia de enteros positivos compuesta de la manera usual a partir de subcadenas disjuntas de D. Por ejemplo, la secuencia 3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279 es una subsecuencia digital de los primeros varios dígitos de π:

3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9

La longitud de una subsecuencia digital es el número de enteros que contiene, no el número de dígitos; el ejemplo anterior tiene longitud 12. Como de costumbre, una subsecuencia digital es creciente si cada número es mayor que su predecesor.

Describe y analiza un algoritmo eficiente para calcular la subsecuencia digital creciente más larga de D. [Pista: Ten cuidado con tus suposiciones computacionales. ¿Cuánto tiempo toma comparar dos números de k dígitos?]

Para crédito completo, tu algoritmo debería funcionar en tiempo O(n⁴); algoritmos más rápidos valen crédito extra. El algoritmo más rápido que conozco para este problema funciona en tiempo O(n^(3/2) log n); lograr esta cota requiere varios trucos, tanto en el diseño del algoritmo como en su análisis, pero nada fuera del alcance de esta clase.²²

♥**31.** Considera la siguiente variante del problema clásico de la Torre de Hanoi. Como de costumbre, hay n discos con tamaños distintos, colocados en tres postes numerados 0, 1, y 2. Inicialmente, todos los n discos están en el poste 0, ordenados por tamaño del más pequeño arriba al más grande abajo. Nuestro objetivo es mover todos los discos al poste 2. En un solo paso, podemos mover el disco más alto en cualquier poste a un poste diferente, siempre que satisfagamos dos restricciones. Primero, nunca debemos colocar un disco más pequeño encima de un disco más grande. Segundo — y esta es la parte no estándar — nunca debemos mover un disco directamente del poste 0 al poste 2.

Describe y analiza un algoritmo para calcular el número exacto de movimientos requeridos para mover todos los n discos del poste 0 al poste 2, sujeto a las restricciones establecidas. Para crédito completo, tu algoritmo debería usar solo O(log n) operaciones aritméticas en el peor caso. Para propósitos de análisis, asume que sumar o multiplicar dos números de k dígitos requiere tiempo O(k). [Pista: ¡Matrices!]

**Dividiendo Secuencias/Arreglos**

**32.** Una expresión aritmética básica está compuesta por caracteres del conjunto {1,+,×} y paréntesis. Casi todo entero puede representarse por más de una expresión aritmética básica. Por ejemplo, todas las siguientes expresiones aritméticas básicas representan el entero 14:

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

((1 + 1) × (1 + 1 + 1 + 1 + 1)) + ((1 + 1) × (1 + 1))

(1 + 1) × (1 + 1 + 1 + 1 + 1 + 1 + 1)

(1 + 1) × (((1 + 1 + 1) × (1 + 1)) + 1)

Describe y analiza un algoritmo para calcular, dado un entero n como entrada, el número mínimo de 1s en una expresión aritmética básica cuyo valor sea igual a n. El número de paréntesis no importa, solo el número de 1s. Por ejemplo, cuando n = 14, tu algoritmo debería retornar 8, para la expresión final de arriba. El tiempo de ejecución de tu algoritmo debería estar acotado por una función polinomial pequeña de n.

**33.** Supón que se te da una secuencia de enteros separados por signos + y −; por ejemplo:

1 + 3 − 2 − 5 + 1 − 6 + 7

Puedes cambiar el valor de esta expresión agregando paréntesis en diferentes lugares. Por ejemplo:

1 + 3 − 2 − 5 + 1 − 6 + 7 = −1

(1 + 3 − (2 − 5)) + (1 − 6) + 7 = 9

(1 + (3 − 2)) − (5 + 1) − (6 + 7) = −17

Describe y analiza un algoritmo para calcular, dada una lista de enteros separados por signos + y −, el valor máximo posible que la expresión puede tomar agregando paréntesis. Los paréntesis deben usarse solo para agrupar sumas y restas; en particular, no los uses para crear multiplicación implícita como en 1 + 3(−2)(−5) + 1 − 6 + 7 = 33.

**34.** Supón que se te da una secuencia de enteros separados por signos + y ×; por ejemplo:

1 + 3 × 2 × 0 + 1 × 6 + 7

Puedes cambiar el valor de esta expresión agregando paréntesis en diferentes lugares. Por ejemplo:

(1 + (3 × 2)) × 0 + (1 × 6) + 7 = 13

((1 + (3 × 2 × 0) + 1) × 6) + 7 = 19

(1 + 3) × 2 × (0 + 1) × (6 + 7) = 104

(a) Describe y analiza un algoritmo para calcular el valor máximo posible que la expresión dada puede tomar agregando paréntesis, asumiendo que todos los enteros en la entrada son positivos. [Pista: Esto es fácil.]

(b) Describe y analiza un algoritmo para calcular el valor máximo posible que la expresión dada puede tomar agregando paréntesis, asumiendo que todos los enteros en la entrada son no negativos.

(c) Describe y analiza un algoritmo para calcular el valor máximo posible que la expresión dada puede tomar agregando paréntesis, sin restricciones en los números de entrada.

Asume que cualquier operación aritmética toma tiempo O(1).

**35.** Después de graduarte de Sham-Poobanana University, decides entrevistarte para una posición en el banco de Wall Street Long Live Boole. El director gerente del banco, Eloob Egroeg, plantea un problema de 'resolver-o-morir' a cada nuevo empleado, que deben resolver en 24 horas. ¡Aquellos que fallan en resolver el problema son despedidos inmediatamente!

Al entrar al banco por primera vez, notas que las oficinas de empleados están organizadas en una fila recta, con una gran T o F impresa en la puerta de cada oficina. Además, entre cada par adyacente de oficinas, hay un tablero marcado por uno de los símbolos ∧, ∨, o ⊕. Cuando preguntas sobre estos símbolos arcanos, Eloob confirma que T y F representan los valores booleanos True y False, y los símbolos en los tableros representan los operadores booleanos estándar And, Or, y Xor. También explica que estas letras y símbolos describen si ciertas combinaciones de empleados pueden trabajar juntos exitosamente. Al inicio de cualquier nuevo proyecto, Eloob agrupa jerárquicamente a sus empleados agregando paréntesis a la secuencia de símbolos, para obtener una expresión booleana inequívoca. El proyecto es exitoso si esta expresión booleana con paréntesis evalúa a T.

Por ejemplo, si el banco tiene tres empleados, y la secuencia de símbolos en y entre sus puertas es T ∧ F ⊕ T, hay exactamente un esquema de parentización exitoso: (T ∧ (F ⊕ T)). Sin embargo, si la lista de símbolos de puertas es F ∧ T ⊕ F, no hay manera de agregar paréntesis para hacer el proyecto exitoso.

Eloob finalmente plantea tu pregunta de entrevista de resolver-o-morir: Describe un algoritmo para decidir si una secuencia dada de símbolos puede parentizarse para que la expresión booleana resultante evalúe a T. Tu entrada es un arreglo S[0 .. 2n], donde S[i] ∈ {T, F} cuando i es par, y S[i] ∈ {∨,∧,⊕} cuando i es impar.

**36.** Cada año, como parte de su reunión anual, los Amantes de Caracoles Antárticos de Upper Glacierville realizan una Carrera de Apareamiento de Mesa Redonda. Varios caracoles reproductores de alta calidad se colocan en el borde de una mesa redonda. Los caracoles están numerados en orden alrededor de la mesa del 1 al n. Durante la carrera, cada caracol vaga por la mesa, dejando un rastro de baba detrás. Los caracoles han sido especialmente entrenados para nunca caerse del borde de la mesa o cruzar un rastro de baba, ni siquiera el suyo propio. Si dos caracoles se encuentran, se declaran una pareja reproductora, se retiran de la mesa, y se los lleva a un agujero romántico en el suelo para hacer pequeños caracoles bebé. Nota que algunos caracoles pueden nunca encontrar pareja, incluso si la carrera continúa para siempre.

**Referencia a Figura 3.6**: El final de una carrera típica de SLUG Antártica. Los caracoles 6 y 8 nunca encuentran parejas. Los organizadores deben pagar M[3, 4] + M[2, 5] + M[1, 7].

Para cada par de caracoles, los organizadores de la carrera SLUG Antártica han publicado una recompensa monetaria, que se pagará a los dueños si ese par de caracoles se encuentra durante la Carrera de Apareamiento. Específicamente, hay un arreglo bidimensional M[1 .. n, 1 .. n] publicado en la pared detrás de la Mesa Redonda, donde M[i, j] = M[j, i] es la recompensa que se pagará si los caracoles i y j se encuentran.

Describe y analiza un algoritmo para calcular la recompensa total máxima que los organizadores podrían verse obligados a pagar, dado el arreglo M como entrada.

**37.** Has extraído una gran losa de mármol de una cantera. Para simplicidad, supón que la losa de mármol es un rectángulo que mide n pulgadas de altura y m pulgadas de ancho. Quieres cortar la losa en rectángulos más pequeños de varios tamaños — algunos para encimeras de cocina, algunos para proyectos de escultura grandes, otros para lápidas conmemorativas. Tienes una sierra de mármol que puede hacer cortes ya sea horizontales o verticales a través de cualquier losa rectangular. En cualquier momento, puedes consultar el precio al contado P[x, y] de un rectángulo de mármol de x pulgadas por y pulgadas, para cualquier enteros positivos x e y. Estos precios dependen de la demanda del cliente, y las personas que compran encimeras de mármol son raras, así que no hagas suposiciones sobre ellos; en particular, rectángulos más grandes pueden tener precios al contado significativamente menores. Dados el arreglo de precios al contado y los enteros m y n como entrada, describe un algoritmo para calcular cómo subdividir una losa de mármol n × m para maximizar tu ganancia.

**38.** Este problema te pide diseñar algoritmos eficientes para construir árboles de búsqueda binaria óptimos que satisfagan restricciones de balance adicionales. Tu entrada consiste en un arreglo ordenado A[1 .. n] de claves de búsqueda y un arreglo f[1 .. n] de conteos de frecuencia, donde f[i] es el número de búsquedas para A[i]. Esta es exactamente la misma función de costo descrita en la Sección 3.9. Pero ahora tu tarea es calcular un árbol óptimo que satisfaga algunas restricciones adicionales.

(a) Los árboles AVL fueron los primeros árboles de búsqueda binaria auto-balanceados, descritos por primera vez en 1962 por Georgy Adelson-Velsky y Evgenii Landis. Un árbol AVL es un árbol de búsqueda binaria donde para cada nodo v, la altura del subárbol izquierdo de v y la altura del subárbol derecho de v difieren por a lo sumo uno.

Describe y analiza un algoritmo para construir un árbol AVL óptimo para un conjunto dado de claves de búsqueda y frecuencias.

(b) Los árboles B binarios simétricos son otros árboles binarios auto-balanceados, descritos por primera vez por Rudolf Bayer en 1972; estos son mejor conocidos por el nombre árboles rojo-negro, después de una reformulación algo más simple por Leo Guibas y Bob Sedgwick en 1978. Un árbol rojo-negro es un árbol de búsqueda binaria con las siguientes restricciones adicionales:

• Cada nodo es ya sea rojo o negro. • Cada nodo rojo tiene un padre negro. • Cada camino de raíz a hoja contiene el mismo número de nodos negros.

Describe un algoritmo recursivo de retroceso para construir un árbol rojo-negro óptimo para un conjunto dado de claves de búsqueda y frecuencias.

(c) Los árboles AA fueron propuestos por Arne Andersson en 1993 y ligeramente simplificados (y nombrados) por Mark Allen Weiss en 2000. Los árboles AA también se conocen como árboles rojo-negro que se inclinan a la izquierda, después de una reformulación simétrica (con algoritmos de rebalanceo diferentes) por Bob Sedgewick en 2006. Un árbol AA es un árbol rojo-negro con una restricción adicional:

• Ningún hijo izquierdo es rojo.²³

Describe y analiza un algoritmo para construir un árbol AA óptimo para un conjunto dado de claves de búsqueda y frecuencias.

**39.** Supón que se te da un mapa de bits m × n como un arreglo M[1 .. m, 1 .. n] de 0s y 1s. Un bloque sólido en M es un subarreglo de la forma M[i .. i′, j .. j′] en el cual todos los bits son iguales. Supón que quieres descomponer M en la menor cantidad posible de bloques disjuntos.

Una estrategia natural de partición recursiva se llama **subdivisión guillotina**. Si todo el mapa de bits M es un bloque sólido, no hay nada que hacer. De otro modo, cortamos M en dos mapas de bits más pequeños a lo largo de una línea horizontal o vertical, y luego descomponemos recursivamente los dos mapas de bits más pequeños en bloques sólidos.

Cualquier subdivisión guillotina puede representarse como un árbol binario, donde cada nodo interno almacena la posición y orientación de un corte, y cada hoja almacena un solo bit 0 o 1 indicando el contenido del bloque correspondiente. El tamaño de una subdivisión guillotina es el número de hojas en el árbol binario correspondiente (es decir, el número final de bloques sólidos), y la profundidad de una subdivisión guillotina es la profundidad del árbol binario correspondiente.

(a) Describe y analiza un algoritmo para calcular una subdivisión guillotina de M del tamaño mínimo posible.

(b) Muestra que una subdivisión guillotina no siempre produce una partición en el menor número de bloques sólidos.

(c) Describe y analiza un algoritmo para calcular una subdivisión guillotina para M con la menor profundidad posible.

(d) Describe y analiza un algoritmo para determinar M[i, j], dado el árbol que representa una descomposición guillotina para M y dos índices i y j.

**Referencia a Figura 3.7**: Una subdivisión guillotina con tamaño 8 y profundidad 5.

(e) Define la profundidad de un píxel M[i, j] en una subdivisión guillotina como la profundidad de la hoja que contiene ese píxel. Describe y analiza un algoritmo para calcular una subdivisión guillotina para M tal que la suma de las profundidades de los píxeles sea lo más pequeña posible.

(f) Describe y analiza un algoritmo para calcular una subdivisión guillotina para M tal que la suma de las profundidades de los píxeles negros sea lo más pequeña posible.

♠**40.** ¡Felicitaciones! Has sido contratado por la librería en línea gigante DeNile ("¡No solo un río en Egipto!") para optimizar sus robots de almacén. Cada libro que DeNile vende tiene un ISBN único (Número Estándar Internacional de Libro), que es solo un valor numérico. Cada uno de los almacenes de DeNile contiene una fila larga de contenedores, cada uno conteniendo múltiples copias de un solo libro. Estos contenedores están organizados en orden ordenado por ISBN; el ISBN de cada contenedor está impreso en el frente del contenedor en forma legible por máquina. Los libros se recuperan de estos contenedores por robots, que corren a lo largo de rieles paralelos a la fila de contenedores.

DeNile no mantiene una lista de qué contenedores contienen qué números ISBN; ¡eso sería demasiado simple! En su lugar, para recuperar un libro deseado, el robot debe primero encontrar el contenedor de ese libro usando una búsqueda binaria. Porque la búsqueda requiere movimiento físico por parte del robot, ya no podemos asumir que cada paso de la búsqueda binaria requiere tiempo O(1). Específicamente:

• El robot siempre comienza en el "contenedor 0" (donde los libros se cargan en cajas para enviar a los clientes). • Mover el robot del i-ésimo contenedor al j-ésimo contenedor requiere α|i − j| segundos para alguna constante α. • El robot debe estar directamente frente a un contenedor para leer el ISBN de ese contenedor. Leer un ISBN requiere β segundos, para alguna constante β. • Revertir la dirección de movimiento del robot (de creciente a decreciente o viceversa) requiere γ segundos adicionales, para alguna constante γ. • Cuando el robot encuentra el contenedor objetivo, extrae un libro de ese contenedor y regresa al "contenedor 0".

Diseña y analiza un algoritmo para calcular un árbol de búsqueda binaria sobre los contenedores que minimice el tiempo total que el robot gasta buscando libros. Tu entrada es un arreglo f[1 .. n] de enteros, donde f[i] es el número de veces que se le pedirá al robot recuperar un libro del i-ésimo contenedor, junto con los parámetros de tiempo α, β, y γ.

♠**41.** Un método estándar para mejorar el rendimiento de caché de árboles de búsqueda es empacar más claves de búsqueda y subárboles en cada nodo. Un B-árbol es un árbol con raíz en el cual cada nodo interno almacena hasta B claves y punteros a hasta B + 1 hijos, cada uno la raíz de un B-árbol más pequeño. Específicamente, cada nodo v almacena tres campos:

• un entero positivo v.d ≤ B, • un arreglo ordenado v.key[1 .. v.d], y • un arreglo v.child[0 .. v.d] de punteros hijo.

En particular, el número de punteros hijo es siempre exactamente uno más que el número de claves.²⁴

Cada puntero v.child[i] es ya sea Null o un puntero a la raíz de un B-árbol cuyas claves son todas mayores que v.key[i] y menores que v.key[i + 1]. En particular, todas las claves en el subárbol más a la izquierda v.child[0] son menores que v.key[1], y todas las claves en el subárbol más a la derecha v.child[v.d] son mayores que v.key[v.d].

Intuitivamente, deberías tener la siguiente imagen en mente:

[ •·⚬ < key[1] < •·⚬ < key[2] < •·⚬ ··· •·⚬ < key[d] < •·⚬ ]

T₀ T₁ T₂ ··· Tₐ₋₁ Tₐ

Aquí Tᵢ es el subárbol apuntado por child[i].

El costo de buscar una clave x en un B-árbol es el número de nodos en el camino desde la raíz hasta el nodo que contiene x como una de sus claves. Un 1-árbol es solo un árbol de búsqueda binaria estándar.

Fija un entero positivo arbitrario B > 0. (Sugiero B = 8.) Supón que se te da un arreglo ordenado A[1, . . . , n] de claves de búsqueda y un arreglo correspondiente F[1, . . . , n] de conteos de frecuencia, donde F[i] es el número de veces que buscaremos A[i]. Tu tarea es describir y analizar un algoritmo eficiente para encontrar un B-árbol que minimice el costo total de buscar las claves dadas con las frecuencias dadas.

(a) Describe un algoritmo de tiempo polinomial para el caso especial B = 2.

(b) Describe un algoritmo para B arbitrario que funciona en tiempo O(n^(B+c)) para algún entero fijo c.

♥(c) Describe un algoritmo para B arbitrario que funciona en tiempo O(n^c) para algún entero fijo c que no depende de B.

**42.** Una cadena w de paréntesis ( y ) y corchetes [ y ] está **balanceada** si satisface una de las siguientes condiciones:

• w es la cadena vacía. • w = (x) para alguna cadena balanceada x • w = [x] para alguna cadena balanceada x • w = x y para algunas cadenas balanceadas x e y

Por ejemplo, la cadena

w = ([()][]()) [()()] ()

está balanceada, porque w = x y, donde

x = ( [()] [] () ) y y = [ () () ] ().

(a) Describe y analiza un algoritmo para determinar si una cadena dada de paréntesis y corchetes está balanceada.

(b) Describe y analiza un algoritmo para calcular la longitud de una subsecuencia balanceada más larga de una cadena dada de paréntesis y corchetes.

(c) Describe y analiza un algoritmo para calcular la longitud de una supersecuencia balanceada más corta de una cadena dada de paréntesis y corchetes.

(d) Describe y analiza un algoritmo para calcular la distancia de edición mínima de una cadena dada de paréntesis y corchetes a una cadena balanceada de paréntesis y corchetes.

♥(e) Describe y analiza un algoritmo para calcular la subsecuencia común balanceada más larga de dos cadenas dadas de paréntesis y corchetes.

♥(f) Describe y analiza un algoritmo para calcular la subsecuencia balanceada palindrómica más larga de una cadena dada de paréntesis y corchetes.

♥(g) Describe y analiza un algoritmo para calcular la subsecuencia común palindrómica balanceada más larga (¡uf!) de dos cadenas dadas de paréntesis y corchetes.

Para cada problema, tu entrada es un arreglo w[1 .. n], donde w[i] ∈ {(, ), [, ]} para cada índice i. (Puedes preferir usar símbolos diferentes en lugar de paréntesis y corchetes — por ejemplo, L, R, l, r o Ã,Â,Ê,É — ¡pero por favor dile a tu calificador qué símbolos estás usando!)

♥**43.** ¡Felicitaciones! Tu equipo de investigación acaba de recibir un proyecto multianual de $50M, financiado conjuntamente por DARPA, Google, y McDonald's, para producir DWIM: ¡El primer compilador que lee la mente de los programadores! Tu propuesta y tus numerosos comunicados de prensa prometen que DWIM corregirá automáticamente errores en cualquier pieza de código dada, mientras modifica ese código lo menos posible. Desafortunadamente, ahora es hora de comenzar a hacer que la maldita cosa funcione.

Como ejercicio de calentamiento, decides abordar el siguiente subproblema necesario. Recuerda que la distancia de edición entre dos cadenas es el número mínimo de inserciones, eliminaciones y reemplazos de caracteres individuales requerido para transformar una cadena en la otra. Una expresión aritmética es una cadena w tal que

• w es una cadena de uno o más dígitos decimales, • w = (x) para alguna expresión aritmética x, o • w = x ⊙ y para algunas expresiones aritméticas x e y y algún operador binario ⊙.

Supón que se te da una cadena de tokens del alfabeto {#, ⊙, (, )}, donde # representa un dígito decimal y ⊙ representa un operador binario. Describe y analiza un algoritmo para calcular la distancia de edición mínima de la cadena dada a una expresión aritmética.

**44.** Las moléculas de ácido ribonucleico (ARN) son cadenas largas de millones de nucleótidos o bases de cuatro tipos diferentes: adenina (A), citosina (C), guanina (G), y uracilo (U). La secuencia de una molécula de ARN es una cadena b[1 .. n], donde cada carácter b[i] ∈ {A, C, G, U} corresponde a una base. Además de los enlaces químicos entre bases adyacentes en la secuencia, los enlaces de hidrógeno pueden formarse entre ciertos pares de bases. El conjunto de pares de bases enlazados se llama la estructura secundaria de la molécula de ARN.

Decimos que dos pares de bases (i, j) y (i′, j′) con i < j e i′ < j′ se superponen si i < i′ < j < j′ o i′ < i < j′ < j. En la práctica, la mayoría de los pares de bases no se superponen. Los pares de bases superpuestos crean los llamados pseudonudos en la estructura secundaria, que son esenciales para algunas funciones del ARN, pero son más difíciles de predecir.

Supón que queremos predecir la mejor estructura secundaria posible para una secuencia de ARN dada. Adoptaremos un modelo drásticamente simplificado de estructura secundaria:

• Cada base puede enlazarse con a lo sumo una otra base. • Solo los pares A–U y C–G pueden enlazarse. • Los pares de la forma (i, i + 1) y (i, i + 2) no pueden enlazarse. • Los pares de bases enlazados no pueden superponerse.

La última (y menos realista) restricción nos permite visualizar la estructura secundaria del ARN como una especie de árbol gordo, como se muestra abajo.

**Referencia a Figura 3.8**: Ejemplo de estructura secundaria de ARN con 21 pares de bases enlazados, indicados por líneas rojas gruesas. Los espacios se indican por curvas punteadas. Esta estructura tiene puntuación 2² + 2² + 8² + 1² + 7² + 4² + 7² = 187.

(a) Describe y analiza un algoritmo que calcule el número máximo posible de pares de bases enlazados en una estructura secundaria para una secuencia de ARN dada.

(b) Un espacio en una estructura secundaria es una subcadena maximal de bases no apareadas. Los espacios grandes llevan a inestabilidades químicas, así que las estructuras secundarias con espacios menores son más probables. Para dar cuenta de esta preferencia, definamos la puntuación de una estructura secundaria como la suma de los cuadrados de las longitudes de los espacios; ver Figura 3.8. (Esta función de puntuación es completamente ficticia; la predicción real de estructura de ARN requiere funciones de puntuación mucho más complicadas.)

Describe y analiza un algoritmo que calcule la puntuación mínima posible de una estructura secundaria para una secuencia de ARN dada.

♣**45.** (a) Describe y analiza un algoritmo eficiente para determinar, dada una cadena w y una expresión regular R, si w ∈ L(R).

(b) Las expresiones regulares generalizadas permiten el operador binario ∩ (intersección) y el operador unario ¬ (complemento), además de los operadores usuales • (concatenación), + (o), y \* (cerradura de Kleene). Las construcciones NFA y el teorema de Kleene implican que cualquier expresión regular generalizada E representa un lenguaje regular L(E).

Describe y analiza un algoritmo eficiente para determinar, dada una cadena w y una expresión regular generalizada E, si w ∈ L(E).

En ambos problemas, asume que en realidad se te da un árbol de análisis para la expresión regular (generalizada), no solo una cadena.

**Ejercicios: Árboles y Subárboles**

**Capítulo 3: Programación Dinámica**

**46. Fiesta Corporativa Obligatoria**

Acabas de ser nombrado como el nuevo organizador de la primera fiesta navideña obligatoria anual en Giggle (una subsidiaria de Abugida). Los empleados de Giggle están organizados en una jerarquía estricta—un árbol con el presidente de la compañía en la raíz. Los oráculos omniscientes de Recursos Humanos han asignado un número real a cada empleado midiendo qué tan "divertido" es el empleado. Para mantener las cosas sociales, hay una restricción en la lista de invitados: un empleado no puede asistir a la fiesta si su supervisor inmediato también está presente. Por otro lado, el presidente de la compañía debe asistir a la fiesta, aunque tenga una calificación de diversión negativa; es su compañía, después de todo. Da un algoritmo que haga una lista de invitados para la fiesta que maximice la suma de las calificaciones de "diversión" de los invitados.

**47. Distribución de Regalos Corporativos**

Dado que tan pocas personas vinieron a la fiesta navideña del año pasado, el presidente de Giggle decide dar a cada empleado un regalo en su lugar este año. Específicamente, cada empleado debe recibir uno de tres regalos: (1) unas vacaciones de seis semanas con todos los gastos pagados en cualquier lugar del mundo, (2) un desayuno de todos-los-panqueques-que-puedas-ordenar para dos en Jumping Jack Flash's Flapjack Stack Shack, o (3) una bolsa de papel ardiendo llena de excremento de perro. Las regulaciones corporativas prohíben que cualquier empleado reciba exactamente el mismo regalo que su supervisor directo. Cualquier empleado que reciba un mejor regalo que su supervisor directo casi seguramente será despedido en un ataque de celos.

Como zar oficial de fiestas de Giggle, es tu trabajo decidir qué regalo recibe cada empleado. Describe un algoritmo para distribuir regalos de manera que el número mínimo de personas sean despedidas. Sí, puedes enviar al presidente una bolsa flamante de excremento de perro.

Más formalmente, se te da un árbol con raíz T, representando la jerarquía de la compañía, y quieres etiquetar los nodos de T con enteros 1, 2, o 3, de manera que cada nodo tenga una etiqueta diferente de su padre. El costo de un etiquetado es el número de nodos con etiquetas menores que sus padres. Ver Figura 3.9 para un ejemplo. Describe y analiza un algoritmo para computar el etiquetado de costo mínimo de T.

**Figura 3.9.** Un etiquetado de árbol con costo 9. Los nueve nodos en negrita tienen etiquetas menores que sus padres. Este no es el etiquetado óptimo para este árbol.

**48. Fiesta de Twitbook con Restricciones**

Después del Desastre Navideño del Excremento de Perro Flamante, fuiste fuertemente alentado a buscar otro empleo, y así dejaste Giggle por la compañía rival Twitbook. Desafortunadamente, el nuevo presidente de Twitbook acababa de decidir imitar a Giggle organizando su propia fiesta navideña, y en vista de tu experiencia pasada, te nombró como el organizador oficial de la fiesta. El presidente exige que invites exactamente k empleados, incluyendo al presidente mismo, y todos los que son invitados están obligados a asistir. Sí, eso será divertido.

Igual que en Giggle, los empleados en Twitbook están organizados en una jerarquía estricta: un árbol con el presidente de la compañía en la raíz. Los oráculos omniscientes de Recursos Humanos han asignado un número real a cada empleado indicando la incomodidad de invitar tanto a ese empleado como a su supervisor inmediato; un valor negativo indica que el empleado y su supervisor realmente se gustan. Tu objetivo es elegir un subconjunto de exactamente k empleados para invitar, de manera que la incomodidad total de la fiesta resultante sea tan pequeña como sea posible. Por ejemplo, si la lista de invitados no incluye tanto a un empleado como a su supervisor inmediato, la incomodidad total es cero. La entrada a tu algoritmo es el árbol T, el entero k, y la incomodidad de cada nodo en T.

**(a)** Describe un algoritmo que compute la incomodidad total del subconjunto menos incómodo de k empleados, asumiendo que la jerarquía de la compañía está descrita por un árbol binario. Es decir, asume que cada empleado supervisa directamente a lo más dos otros.

**♥(b)** Describe un algoritmo que compute la incomodidad total del subconjunto menos incómodo de k empleados, sin restricciones en la jerarquía de la compañía.

**49. Difusión de Mensajes en Árboles**

Supón que necesitamos difundir un mensaje a todos los nodos en un árbol con raíz. Inicialmente, solo el nodo raíz conoce el mensaje. En una sola ronda, cualquier nodo que conoce el mensaje puede reenviarlo a lo más uno de sus hijos. Ver Figura 3.10 para un ejemplo.

**(a)** Diseña un algoritmo para computar el número mínimo de rondas requeridas para difundir el mensaje a todos los nodos en un árbol binario.

**♦(b)** Diseña un algoritmo para computar el número mínimo de rondas requeridas para difundir el mensaje a todos los nodos en un árbol con raíz arbitrario. [Pista: Puedes encontrar útiles las técnicas del siguiente capítulo para demostrar que tu algoritmo es correcto, aunque no sea un algoritmo codicioso.]

**Figura 3.10.** Un mensaje siendo distribuido a través de un árbol en cinco rondas.

**50. Juego de Escalada en Roca**

Un día, Alex se cansó de escalar en un gimnasio y decidió llevar a un grupo muy grande de amigos escaladores afuera a escalar. El área de escalada donde fueron, tenía una enorme roca ancha, no muy alta, con varios agarres marcados para manos y pies. Alex rápidamente determinó un conjunto "permitido" de movimientos que su grupo de amigos puede realizar para ir de un agarre a otro.

El sistema general de agarres puede describirse por un árbol con raíz T con n vértices, donde cada vértice corresponde a un agarre y cada arista corresponde a un movimiento permitido entre agarres. Los caminos de escalada convergen conforme van hacia arriba de la roca, llevando a un agarre único en la cumbre, representado por la raíz de T.

Alex y sus amigos (que son todos escaladores excelentes) decidieron jugar un juego, donde tantos escaladores como sea posible están simultáneamente en la roca y cada escalador necesita realizar una secuencia de exactamente k movimientos. Cada escalador puede elegir un agarre arbitrario para comenzar, y todos los movimientos deben moverse alejándose del suelo. Así, cada escalador traza un camino de k aristas en el árbol T, todas dirigidas hacia la raíz. Sin embargo, no se permite que dos escaladores toquen el mismo agarre; los caminos seguidos por diferentes escaladores no pueden intersectarse en absoluto.

Describe y analiza un algoritmo eficiente para computar el número máximo de escaladores que pueden jugar este juego. Más formalmente, se te da un árbol con raíz T y un entero k, y quieres encontrar el mayor número posible de caminos disjuntos en T, donde cada camino tiene longitud k. No asumas que T es un árbol binario. Por ejemplo, dado el árbol T abajo y k = 3 como entrada, tu algoritmo debería retornar el entero 8.²⁵

**Figura 3.11.** Siete caminos disjuntos de longitud k = 3. Este no es el conjunto más grande de tales caminos en este árbol.

**51. Costo de Agrupamiento en Árboles Binarios**

Sea T un árbol binario con raíz con n vértices, y sea k ≤ n un entero positivo. Nos gustaría marcar k vértices en T de manera que cada vértice tenga un ancestro marcado cercano. Más formalmente, definimos el costo de agrupamiento de cualquier subconjunto K de vértices como

cost(K) = max\_v cost(v, K),

donde el máximo se toma sobre todos los vértices v en el árbol, y cost(v, K) es la distancia de v a su ancestro más cercano en K:

cost(v, K) = { 0 si v ∈ K ∞ si v es la raíz de T y v ∉ K  
1 + cost(parent(v), K) en otro caso }

En particular, cost(K) = ∞ si K excluye la raíz de T.

**Figura 3.12.** Un subconjunto de cinco vértices en un árbol binario, con costo de agrupamiento 3.

**♥(a)** Describe un algoritmo de programación dinámica para computar, dado el árbol T y un entero k, el costo mínimo de agrupamiento de cualquier subconjunto de k vértices en T. Para crédito completo, tu algoritmo debería ejecutar en tiempo O(n²k²).

**(b)** Describe un algoritmo de programación dinámica para computar, dado el árbol T y un entero r, el tamaño del subconjunto más pequeño de vértices cuyo costo de agrupamiento es a lo más r. Para crédito completo, tu algoritmo debería ejecutar en tiempo O(nr).

**(c)** Muestra que tu solución para la parte (b) implica un algoritmo para la parte (a) que ejecuta en tiempo O(n² log n).

**52. Subárboles Comunes con Raíz**

Esta pregunta te pide encontrar algoritmos eficientes para computar el subárbol común con raíz más grande de dos árboles con raíz dados. Recuerda que un árbol con raíz es un grafo acíclico conectado con un nodo designado llamado la raíz. Un subárbol con raíz de un árbol con raíz consiste en un nodo arbitrario y todos sus descendientes. La definición precisa de "común" depende de qué pares de árboles con raíz consideramos isomórficos.

**(a)** Recuerda que un árbol binario es un árbol con raíz en el cual cada nodo tiene un subárbol izquierdo (posiblemente vacío) y un subárbol derecho (posiblemente vacío). Dos árboles binarios son isomórficos si y solo si ambos están vacíos, o sus subárboles izquierdos son isomórficos y sus subárboles derechos son isomórficos. Describe un algoritmo para encontrar el subárbol binario común más grande de dos árboles binarios dados.

**Figura 3.13.** Dos árboles binarios, con su subárbol común (con raíz) más grande enfatizado.

**(b)** En un árbol con raíz ordenado, cada nodo tiene una secuencia de hijos, que son las raíces de subárboles con raíz ordenados. Dos árboles con raíz ordenados son isomórficos si ambos están vacíos, o si sus i-ésimos subárboles son isomórficos para cada índice i. Describe un algoritmo para encontrar el subárbol ordenado común más grande de dos árboles ordenados T₁ y T₂.

**♦♥(c)** En un árbol con raíz no ordenado, cada nodo tiene un conjunto no ordenado de hijos, que son las raíces de subárboles con raíz no ordenados. Dos árboles con raíz no ordenados son isomórficos si ambos están vacíos, o los subárboles de cada raíz pueden ordenarse de manera que sus i-ésimos subárboles sean isomórficos para cada índice i. Describe un algoritmo para encontrar el subárbol no ordenado común más grande de dos árboles no ordenados T₁ y T₂.

**53. Subárboles Óptimos en Árboles Sin Raíz**

Esta pregunta te pide encontrar algoritmos eficientes para computar subárboles óptimos en árboles sin raíz—grafos no dirigidos acíclicos conectados. Un subárbol de un árbol sin raíz es cualquier subgrafo conectado.

**(a)** Supón que se te da un árbol sin raíz T con pesos en sus aristas, que pueden ser positivos, negativos, o cero. Describe un algoritmo para encontrar un camino en T con peso total máximo.

**(b)** Supón que se te da un árbol sin raíz T con pesos en sus vértices, que pueden ser positivos, negativos, o cero. Describe un algoritmo para encontrar un subárbol de T con peso total máximo. [Esta fue una pregunta de entrevista de Google de 2016.]

**(c)** Sean T₁ y T₂ árboles sin raíz ordenados arbitrarios, significando que los vecinos de cada nodo tienen un orden cíclico bien definido. Describe un algoritmo para encontrar el subárbol ordenado común más grande de T₁ y T₂.

**♦♥(d)** Sean T₁ y T₂ árboles sin raíz no ordenados arbitrarios. Describe un algoritmo para encontrar el subárbol no ordenado común más grande de T₁ y T₂.

**54. Menores con Raíz de Árboles con Raíz**

Los menores con raíz de árboles con raíz son una generalización natural de las subsecuencias. Un menor con raíz de un árbol con raíz T es cualquier árbol obtenido contrayendo una o más aristas. Cuando contraemos una arista uv, donde u es el padre de v, los hijos de v se convierten en nuevos hijos de u y luego v es eliminado. En particular, la raíz de T es también la raíz de cada menor con raíz de T.

**Figura 3.14.** Un árbol con raíz y uno de sus menores con raíz.

**(a)** Sea T un árbol con raíz con nodos etiquetados. Decimos que T es aburrido si, para cada nodo x, todos los hijos de x tienen la misma etiqueta; hijos de diferentes nodos pueden tener etiquetas diferentes. Describe un algoritmo para encontrar el menor con raíz aburrido más grande de un árbol con raíz etiquetado dado.

**(b)** Supón que se nos da un árbol con raíz T cuyos nodos están etiquetados con números. Describe un algoritmo para encontrar el menor con raíz ordenado como heap más grande de T. Es decir, tu algoritmo debería retornar el menor con raíz más grande M tal que cada nodo en M tiene una etiqueta menor que sus hijos en M.

**c)** Supón que se nos da un árbol binario T cuyos nodos están etiquetados con números. Describe un algoritmo para encontrar el menor con raíz ordenado como árbol de búsqueda binaria más grande de T. Es decir, tu algoritmo debería retornar un menor con raíz M tal que cada nodo en M tiene a lo más dos hijos, y un recorrido inorden de M es una subsecuencia creciente de un recorrido inorden de T.

**(d)** Recuerda que un árbol con raíz es ordenado si los hijos de cada nodo tienen un orden izquierda-a-derecha bien definido. Describe un algoritmo para encontrar el menor ordenado como árbol de búsqueda binaria más grande de un árbol ordenado arbitrario T cuyos nodos están etiquetados con números. Nuevamente, el orden izquierda-a-derecha de los nodos en M debería ser consistente con su orden en T.

**♥(e)** Describe un algoritmo para encontrar el menor con raíz ordenado común más grande de dos árboles con raíz ordenados etiquetados.

**♦♥(f)** Describe un algoritmo para encontrar el menor con raíz no ordenado común más grande de dos árboles con raíz no ordenados etiquetados. [Pista: Combina programación dinámica con flujos máximos.]

**Notas al Pie**

²⁵ P: ¿Por qué los profesores de ciencias de la computación piensan que los árboles tienen sus raíces en la parte superior? R: ¡Porque nunca han estado afuera!

**Notas al Pie Consolidadas**

**Notas al Pie del Libro "Algoritmos" - Jeff Erickson**

¹ En código Morse, un "dah" dura tres veces más que un "dit", pero cada "dit" o "dah" es seguido por una pausa con la misma duración que un "dit". Así, cada "dit-pausa" es un laghu akṣara, cada "dah-pausa" es un guru akṣara, y hay exactamente cinco letras (M, D, R, U, y H) cuyos códigos duran cuatro mātrā.

² El Chandaḥśāstra contiene dos reglas sistemáticas para listar todos los metros con un número dado de sílabas, que corresponden aproximadamente a escribir números en binario de izquierda a derecha (como los griegos) o de derecha a izquierda (como los egipcios). El mismo texto incluye un algoritmo recursivo para calcular 2ⁿ (el número de metros con n sílabas) por elevación al cuadrado repetida, y (posiblemente) un algoritmo recursivo para calcular coeficientes binomiales (el número de metros con k sílabas cortas y n sílabas en total).

³ "Ningún descubrimiento científico es nombrado por su descubridor original." En su artículo de 1980 que da nombre a la ley, el estadístico Stephen Stigler bromeó que esta ley fue propuesta por primera vez por el sociólogo Robert K. Merton. Sin embargo, declaraciones similares fueron hechas previamente por Vladimir Arnol'd en los años 1970 ("Los descubrimientos raramente se atribuyen a la persona correcta."), Carl Boyer en 1968 ("¡Clio, la musa de la historia, a menudo es voluble al adjuntar nombres a teoremas!"), Alfred North Whitehead en 1917 ("Todo lo que es importante ha sido dicho antes por alguien que no lo descubrió."), e incluso el padre de Stephen, George Stigler en 1966 ("Si alguna vez encontráramos un caso donde una teoría sea nombrada por el hombre correcto, será notado."). Veremos muchos otros ejemplos de la ley de Stigler en este libro.

⁴ Ver http://algorithms.wtf para notas sobre resolver recurrencias de retroceso.

⁵ Michie propuso que los lenguajes de programación deberían soportar una abstracción que llamó "función memo", consistiendo tanto en una función estándar ("regla") como en un diccionario ("rutina"), en lugar de soportar separadamente arreglos y funciones. Cuando una función memo calcula un valor de función por primera vez, "memoriza" (sí, con una R) ese valor en su diccionario. Michie fue inspirado por el uso de Samuel de "aprendizaje rutinario" para acelerar la evaluación recursiva de árboles de juego de damas; Michie describe su propuesta más general como "permitir al programador 'Samuelizar' cualquier función que le plazca." (Por lo que puedo decir, Michie nunca usó el término "memoización" él mismo.) La memoización fue usada incluso antes por el robot resolvedor de laberintos "Teseo" de Claude Shannon, que diseñó y construyó en 1950.

⁶ Técnicas más generales de programación dinámica fueron desplegadas independientemente varias veces a finales de los años 1930 y principios de los 1940. Por ejemplo, Pierre Massé usó algoritmos de programación dinámica para optimizar la operación de presas hidroeléctricas en Francia durante el régimen de Vichy. John von Neumann y Oskar Morgenstern desarrollaron algoritmos de programación dinámica para determinar el ganador de cualquier juego de dos jugadores con información perfecta (por ejemplo, damas). Alan Turing y sus cohorts usaron métodos similares como parte de sus esfuerzos de ruptura de códigos en Bletchley Park. Tanto el trabajo de Massé como el de von Neumann y Morgenstern fueron publicados por primera vez en 1944, seis años antes de que Bellman acuñara la frase "programación dinámica". Los detalles del "Banburismus" de Turing se mantuvieron en secreto hasta mediados de los años 1980.

⁷ Charles Erwin Wilson se convirtió en Secretario de Defensa en enero de 1953, después de una docena de años como presidente de General Motors. "Engine Charlie" reorganizó el Departamento de Defensa y disminuyó significativamente su presupuesto en su primer año en el cargo, con el objetivo explícito de dirigir el Departamento mucho más como una corporación industrial. Bellman describió a Wilson en su autobiografía de 1984 de la siguiente manera:

Teníamos un caballero muy interesante en Washington llamado Wilson. Era secretario de Defensa, y en realidad tenía un miedo y odio patológico de la palabra "investigación". No estoy usando el término a la ligera; lo estoy usando precisamente. Su cara se ponía roja, se volvía rojo, y se ponía violento si la gente usaba el término "investigación" en su presencia. Puedes imaginar cómo se sentía, entonces, sobre el término "matemático"... Sentí que tenía que hacer algo para proteger a Wilson y la Fuerza Aérea del hecho de que realmente estaba haciendo matemáticas dentro de la Corporación RAND. ¿Qué título, qué nombre, podría elegir?

Sin embargo, el primer uso publicado por Bellman del término "programación dinámica" ya apareció en 1952, varios meses antes de que Wilson tomara el cargo, así que esta historia está al menos ligeramente embellecida.

⁸ ... y posiblemente una variación del icónico nombre de marca "Dynamic-Tension" para la famosa serie de ejercicios de Charles Atlas, que Charles Roman acuñó en 1928. ¡Héroe de la Playa!

⁹ como sugirió Piṅgala para potencias de 2 en otra parte del Chandaḥśāstra

¹⁰ ¡Casi nunca funcionan! Entonces da tres hurras, y un hurra más, para el riguroso Capitán del Pinafore! ¡Entonces da tres hurras, y un hurra más, para el Capitán del Pinafore!

¹¹ Introducir esta política en mis propios cursos de algoritmos mejoró significativamente las calificaciones de los estudiantes, porque redujo significativamente la frecuencia de algoritmos codiciosos incorrectos.

¹² De hecho, solo necesitamos la mitad de este arreglo, porque siempre tenemos i < j. Pero incluso si nos importaran los factores constantes en este libro (no nos importan), este sería el momento equivocado para preocuparse por ellos. Primero haz que funcione; luego hazlo mejor.

¹³ ¿Ves?, ¡te dije que no te preocuparas por los factores constantes todavía!

¹⁴ Este algoritmo a veces también se atribuye incorrectamente a Saul Needleman y Christian Wunsch en 1970. "El algoritmo de Needleman-Wunsch" más comúnmente se refiere al algoritmo estándar de programación dinámica para calcular la subsecuencia común más larga de dos cadenas (o equivalentemente, la distancia de edición donde solo se permiten inserciones y eliminaciones) en tiempo O(mn), pero ¡esa atribución también es incorrecta! De hecho, el algoritmo de Needleman y Wunsch calcula subsecuencias comunes más largas (ponderadas) (posiblemente con costos de espacio) en tiempo O(m²n²), usando una recurrencia diferente. Sankoff explícitamente describe su algoritmo de tiempo O(mn) como una mejora del algoritmo de Needleman y Wunsch.

¹⁵ Sí, estoy rompiendo mi propia regla contra la optimización prematura.

¹⁶ Aunque el problema de suma de subconjuntos es NP-difícil, esta cota de tiempo no implica que P=NP, porque T no está necesariamente acotado por una función polinomial del tamaño de entrada.

¹⁷ En el memorándum de investigación de 1967 (!) donde propuso funciones memo, Donald Michie escribió, "Tabular valores de una función que no serán necesarios es un desperdicio de espacio, y recomputar los mismos valores más de una vez es un desperdicio de tiempo." ¡Pero de hecho, tabular valores de una función que no serán necesarios también es un desperdicio de tiempo!

¹⁸ En mis clases de algoritmos, cualquier solución de programación dinámica que no incluya una especificación en inglés de los subproblemas recursivos subyacentes automáticamente obtiene una puntuación de cero, incluso si la solución es por lo demás perfecta. Introducir esta política mejoró significativamente las calificaciones de los estudiantes, porque redujo significativamente el número de veces que enviaron algoritmos de programación dinámica incorrectos (o incoherentes).

¹⁹ Para más detalles sobre la historia y cultura de Nadiria, incluyendo imágenes de las varias denominaciones de Dream-Dollars, ver http://moneyart.biz/dd/.

²⁰ La ladera norte es más rápida, pero demasiado corta para un concurso interesante.

²¹ Si también permitiéramos movimiento hacia arriba, el juego resultante...

²² Con técnicas más avanzadas, creo que el tiempo de ejecución puede reducirse a O(n^(3/2) loglog n), pero no he trabajado los detalles.

²³ La reformulación de Sedgwick requiere que ningún hijo derecho sea rojo. Como sea. Andersson y Sedgwick son extrañamente silenciosos sobre si medir ángulos en sentido horario o antihorario, si Plutón es un planeta, si "rango menor" significa "mejor" o "peor", y si es mejor luchar contra cien caballos del tamaño de patos o un solo pato del tamaño de un caballo.

²⁴ Normalmente, los B-árboles están obligados a satisfacer dos restricciones adicionales, que garantizan un costo de búsqueda en el peor caso de O(log\_B n): Cada hoja debe tener exactamente la misma profundidad, y cada nodo excepto posiblemente la raíz debe contener al menos B/2 claves. Sin embargo, en este problema, no estamos interesados en optimizar el costo de búsqueda en el peor caso, sino más bien el costo total de una secuencia de búsquedas, así que no impondremos estas restricciones adicionales.

²⁵ P: ¿Por qué los profesores de ciencias de la computación piensan que los árboles tienen sus raíces en la parte superior? R: ¡Porque nunca han estado afuera!

**Nota:** Estas notas al pie han sido consolidadas de múltiples capítulos del libro "Algoritmos" de Jeff Erickson y traducidas al español. Cada nota mantiene su numeración original y contexto académico.

**Ejercicios**

**(c)** Supongamos que se nos da un árbol binario T cuyos nodos están etiquetados con números. Describe un algoritmo para encontrar el menor arraigado ordenado por búsqueda binaria más grande de T. Es decir, tu algoritmo debe devolver un menor arraigado M tal que cada nodo en M tenga como máximo dos hijos, y un recorrido inorden de M sea una subsecuencia creciente de un recorrido inorden de T.

**(d)** Recuerda que un árbol arraigado está ordenado si los hijos de cada nodo tienen un orden izquierda-derecha bien definido. Describe un algoritmo para encontrar el menor ordenado por búsqueda binaria más grande de un árbol ordenado arbitrario T cuyos nodos están etiquetados con números. Nuevamente, el orden izquierda-derecha de los nodos en M debe ser consistente con su orden en T.

**♥(e)** Describe un algoritmo para encontrar el menor común arraigado ordenado más grande de dos árboles arraigados etiquetados ordenados.

**♦♥(f)** Describe un algoritmo para encontrar el menor común arraigado no ordenado más grande de dos árboles arraigados etiquetados no ordenados. [Pista: Combina programación dinámica con flujos máximos.]

*El punto es, damas y caballeros, que la codicia es buena. La codicia funciona, la codicia es correcta. La codicia clarifica, corta y captura la esencia del espíritu evolutivo. La codicia en todas sus formas, codicia por la vida, el dinero, el amor, el conocimiento ha marcado el auge hacia arriba de la humanidad. Y la codicia—marca mis palabras—no solo salvará Teldar Paper sino la otra corporación mal funcionando llamada los EUA.* — Gordon Gekko [Michael Douglas], Wall Street (1987)

*Siempre hay una solución fácil para cada problema humano—ordenada, plausible y equivocada.* — H. L. Mencken, "The Divine Afflatus", New York Evening Mail (16 de noviembre de 1917)

**4. Algoritmos Greedy**

**4.1 Almacenando Archivos en Cinta**

Supongamos que tenemos un conjunto de n archivos que queremos almacenar en cinta magnética.¹ En el futuro, los usuarios querrán leer esos archivos de la cinta. Leer un archivo de cinta no es como leer un archivo del disco; primero tenemos que avanzar rápidamente pasando todos los otros archivos, y eso toma una cantidad significativa de tiempo. Sea L[1 .. n] un arreglo que lista las longitudes de cada archivo; específicamente, el archivo i tiene longitud L[i]. Si los archivos se almacenan en orden del 1 al n, entonces el costo de acceder al k-ésimo archivo es

costo(k) = Σᵢ₌₁ᵏ L[i].

¹ Los lectores que estén tentados a objetar que la cinta magnética ha sido obsoleta por décadas están cordialmente invitados a visitar su instalación de supercomputación más cercana; pregunta por ver los robots de cinta. Alternativamente, considera archivar una secuencia de libros en una estantería de biblioteca. Ya sabes, esos extraños objetos como ladrillos hechos de árboles muertos y tinta.

El costo refleja el hecho de que antes de leer el archivo k primero debemos escanear pasando todos los archivos anteriores en la cinta. Si asumimos por el momento que cada archivo es igualmente probable de ser accedido, entonces el costo esperado de buscar un archivo aleatorio es

E[costo] = Σₖ₌₁ⁿ costo(k)/n = (1/n) Σₖ₌₁ⁿ Σᵢ₌₁ᵏ L[i].

Si cambiamos el orden de los archivos en la cinta, cambiamos el costo de acceder a los archivos; algunos archivos se vuelven más caros de leer, pero otros se vuelven más baratos. Diferentes órdenes de archivos probablemente resulten en diferentes costos esperados. Específicamente, sea π(i) el índice del archivo almacenado en la posición i en la cinta. Entonces el costo esperado de la permutación π es

E[costo(π)] = (1/n) Σₖ₌₁ⁿ Σᵢ₌₁ᵏ L[π(i)].

¿Qué orden deberíamos usar si queremos que este costo esperado sea lo más pequeño posible? La respuesta parece intuitivamente clara: Ordenar los archivos por longitud creciente. ¡Pero la intuición es una bestia traicionera. La única manera de estar seguros de que este orden funciona es despegar y bombardear todo el sitio desde la órbita en realidad probar que funciona!

**Lema 4.1.** E[costo(π)] se minimiza cuando L[π(i)] ≤ L[π(i + 1)] para todo i.

**Demostración:** Supón que L[π(i)] > L[π(i + 1)] para algún índice i. Para simplificar la notación, sea a = π(i) y b = π(i + 1). Si intercambiamos los archivos a y b, entonces el costo de acceder a a aumenta por L[b], y el costo de acceder a b disminuye por L[a]. En general, el intercambio cambia el costo esperado por (L[b] − L[a])/n. Pero este cambio es una mejora, porque L[b] < L[a]. Por lo tanto, si los archivos están fuera de orden, podemos disminuir el costo esperado intercambiando algún par de archivos mal ordenados. □

Este es nuestro primer ejemplo de un algoritmo greedy correcto. Para minimizar el costo total esperado de acceder a los archivos, ponemos el archivo que es más barato de acceder primero, y luego escribimos recursivamente todo lo demás; sin retroceso, sin programación dinámica, solo hacer la mejor elección local y avanzar ciegamente. Si usamos un algoritmo de ordenamiento eficiente, el tiempo de ejecución es claramente O(n log n), más el tiempo requerido para escribir realmente los archivos. Para mostrar que el algoritmo greedy es realmente correcto, probamos que la salida de cualquier otro algoritmo puede ser mejorada por algún tipo de intercambio.

Generalicemos esta idea más. Supón que también se nos da un arreglo F[1 .. n] de frecuencias de acceso para cada archivo; el archivo i será accedido exactamente F[i] veces durante la vida útil de la cinta. Ahora el costo total de acceder a todos los archivos en la cinta es

Σcosto(π) = Σₖ₌₁ⁿ F[π(k)] · Σᵢ₌₁ᵏ L[π(i)] = Σₖ₌₁ⁿ Σᵢ₌₁ᵏ F[π(k)] · L[π(i)].

Como antes, reordenar los archivos puede cambiar este costo total. Entonces, ¿qué orden deberíamos usar si queremos que el costo total sea lo más pequeño posible? (Esta pregunta es similar en espíritu al problema del árbol de búsqueda binaria óptimo, pero la estructura de datos objetivo y la función de costo son ambas diferentes, así que el algoritmo debe ser diferente también.)

Ya probamos que si todas las frecuencias son iguales, deberíamos ordenar los archivos por tamaño creciente. Si las frecuencias son todas diferentes pero las longitudes de archivo L[i] son todas iguales, entonces intuitivamente, deberíamos ordenar los archivos por frecuencia de acceso decreciente, con el archivo más accedido primero. De hecho, esto no es difícil de probar (pista, pista) modificando la demostración del Lema 4.1. Pero ¿qué pasa si los tamaños y las frecuencias varían ambos? En este caso, deberíamos ordenar los archivos por la razón L/F.

**Lema 4.2.** Σcosto(π) se minimiza cuando L[π(i)]/F[π(i)] ≤ L[π(i + 1)]/F[π(i + 1)] para todo i.

**Demostración:** Supón que L[π(i)]/F[π(i)] > L[π(i + 1)]/F[π(i + 1)] para algún índice i. Para simplificar la notación, sea a = π(i) y b = π(i+1). Si intercambiamos los archivos a y b, entonces el costo de acceder a a aumenta por L[b], y el costo de acceder a b disminuye por L[a]. En general, el intercambio cambia el costo total por L[b]F[a] − L[a]F[b]. Pero este cambio es una mejora, porque

L[a]/F[a] > L[b]/F[b] ⟺ L[b]F[a] − L[a]F[b] < 0.

Por lo tanto, si dos archivos adyacentes cualesquiera están fuera de orden, podemos mejorar el costo total intercambiándolos. □

**4.2 Programación de Clases**

El siguiente ejemplo es ligeramente más complejo. Supón que decides abandonar las ciencias de la computación y cambiar tu especialidad a Caos Aplicado. El departamento de Caos Aplicado ofrece todas sus clases el mismo día cada semana, llamado "Día Sobrio" por los estudiantes (pero curiosamente, no por la facultad). Cada clase tiene una hora de inicio diferente y una hora de finalización diferente: AC 101 ("Arquitectura del Paisaje de Papel Higiénico") comienza a las 10:27pm y termina a las 11:51pm; AC 666 ("Inmanentizando el Escatón") comienza a las 4:18pm y termina a las 4:22pm, y así sucesivamente.

En interés de graduarse lo más rápido posible, quieres registrarte para tantas clases como sea posible. (Las clases de Caos Aplicado no requieren trabajo real.) La computadora de registro de la universidad no te permite registrarte para clases que se superpongan, y nadie en el departamento sabe cómo anular esta "característica". ¿Qué clases deberías tomar?

Más formalmente, supón que se te dan dos arreglos S[1 .. n] y F[1 .. n] listando las horas de inicio y finalización de cada clase; para ser concretos, podemos asumir que 0 ≤ S[i] < F[i] ≤ M para cada i, para algún valor M (por ejemplo, el número de picosegundos en el Día Sobrio). Tu tarea es elegir el subconjunto más grande posible X ∈ {1, 2, . . . , n} tal que para cualquier par i, j ∈ X, ya sea S[i] > F[j] o S[j] > F[i].

Podemos ilustrar el problema dibujando cada clase como un rectángulo cuyas coordenadas x izquierda y derecha muestran las horas de inicio y finalización. El objetivo es encontrar el subconjunto más grande de rectángulos que no se superpongan verticalmente.

**Figura 4.1.** Un horario libre de conflictos máximo para un conjunto de clases.

Este problema tiene una solución recursiva bastante simple, basada en la observación de que o tomas la clase 1 o no la tomas. Sea B el conjunto de clases que terminan antes de que la clase 1 comience, y sea A el conjunto de clases que comienzan después de que la clase 1 termine:

B := {i | 2 ≤ i ≤ n y F[i] < S[1]} A := {i | 2 ≤ i ≤ n y S[i] > F[1]}

Si la clase 1 está en el horario óptimo, entonces también lo están los horarios óptimos para B y A, que podemos encontrar recursivamente. Si no, podemos encontrar el horario óptimo para {2, 3, . . . , n} recursivamente. Así que deberíamos probar ambas elecciones y tomar la que dé el mejor horario. Evaluar este algoritmo recursivo de abajo hacia arriba nos da un algoritmo de programación dinámica que ejecuta en tiempo O(n³). No me molestaré en repasar los detalles, porque podemos hacerlo mejor.²

Intuitivamente, nos gustaría que la primera clase termine lo más temprano posible, porque eso nos deja con el mayor número de clases restantes. Esta intuición sugiere el siguiente algoritmo greedy simple. Escanea a través de las clases en orden de tiempo de finalización; cuando encuentres una clase que no conflicte con tu última clase hasta ahora, ¡tómala! Ve la Figura 4.2 para una visualización del horario greedy resultante.

Podemos escribir el algoritmo greedy algo más formalmente como se muestra en la Figura 4.3. (Esperemos que la primera línea sea comprensible.) Después del ordenamiento inicial, el algoritmo es un bucle lineal simple, así que todo el algoritmo ejecuta en tiempo O(n log n).

² Pero aún deberías trabajar los detalles tú mismo. El algoritmo de programación dinámica puede usarse para encontrar el "mejor" horario para varias definiciones diferentes de "mejor", pero el algoritmo greedy que estoy describiendo aquí solo funciona cuando "mejor" significa "más grande". También, puedes mejorar el tiempo de ejecución a O(n²) usando una recurrencia diferente.

**Figura 4.2.** Las mismas clases ordenadas por tiempos de finalización y el horario greedy.

**GreedySchedule(S[1 .. n], F[1 .. n]):**

ordenar F y permutar S para que coincida

count ← 1

X[count] ← 1

for i ← 2 to n

if S[i] > F[X[count]]

count ← count + 1

X[count] ← i

return X[1 ..count]

**Figura 4.3.** Un algoritmo greedy para encontrar un conjunto máximo de clases que no se superpongan

Para probar que GreedySchedule realmente calcula el horario libre de conflictos más grande, usamos un argumento de intercambio, similar al que usamos para el ordenamiento de cintas. No estamos afirmando que el horario greedy es el único horario máximo; podría haber otros. (¡Compara las Figuras 4.1 y 4.2!) Todo lo que podemos afirmar es que al menos uno de los horarios óptimos es el producido por el algoritmo greedy.

**Lema 4.3.** Al menos un horario libre de conflictos máximo incluye la clase que termina primero.

**Demostración:** Sea f la clase que termina primero. Supón que tenemos un horario libre de conflictos máximo X que no incluye f. Sea g la primera clase en X que termina. Dado que f termina antes que g, f no puede entrar en conflicto con ninguna clase en el conjunto X \ {g}. Por lo tanto, el horario X' = X ∪ {f} \ {g} también está libre de conflictos. Dado que X' tiene el mismo tamaño que X, también es máximo. □

Para terminar la demostración, llamamos a nuestro viejo amigo la inducción.

**Teorema 4.4.** El horario greedy es un horario óptimo.

**Demostración:** Sea f la clase que termina primero, y sea A el subconjunto de clases que comienzan después de que f termine. El lema anterior implica que algún horario óptimo contiene f, así que el mejor horario que contiene f es un horario óptimo. El mejor horario que incluye f debe contener un horario óptimo para las clases que no entran en conflicto con f, es decir, un horario óptimo para A. El algoritmo greedy elige f y luego, por la hipótesis inductiva, calcula un horario óptimo de clases de A. □

La demostración podría ser más fácil de entender si desenrollamos la inducción ligeramente.

**Demostración:** Sea ⟨g₁, g₂, . . . , gₖ⟩ la secuencia de clases elegidas por el algoritmo greedy, ordenadas por hora de inicio. Supón que tenemos un horario libre de conflictos máximo

S = ⟨g₁, g₂, . . . , gⱼ₋₁, cⱼ, cⱼ₊₁, . . . , cₘ⟩,

nuevamente ordenado por hora de inicio, donde cⱼ es diferente de la clase gⱼ elegida por el algoritmo greedy. (Podríamos tener j = 1, en cuyo caso este horario comienza con una elección no greedy c₁.) Por construcción, la j-ésima elección greedy gⱼ no entra en conflicto con ninguna clase anterior g₁, g₂, . . . , gⱼ₋₁, y porque nuestro horario S está libre de conflictos, tampoco lo hace cⱼ. Además, gⱼ tiene el tiempo de finalización más temprano entre todas las clases que no entran en conflicto con las clases anteriores; en particular, gⱼ termina antes que cⱼ. Se sigue que gⱼ no entra en conflicto con ninguna de las clases posteriores cⱼ₊₁, . . . , cₘ. Por lo tanto, el horario modificado

S' = ⟨g₁, g₂, . . . , gⱼ₋₁, gⱼ, cⱼ₊₁, . . . , cₘ⟩,

también está libre de conflictos. (Este argumento es una generalización directa del Lema 4.3, que considera el caso j = 1.)

Por inducción, ahora se sigue que hay un horario óptimo ⟨g₁, g₂, . . . , gₖ, cₖ₊₁, . . . , cₘ⟩ que incluye cada clase elegida por el algoritmo greedy. Pero esto es imposible a menos que k = m; si alguna clase cₖ₊₁ no entra en conflicto con ninguna de las primeras k clases greedy, ¡entonces el algoritmo greedy elegiría más de k clases! □

**4.3 Patrón General**

La estructura básica de esta demostración de corrección es exactamente la misma que para el problema de ordenamiento de cintas: un argumento de intercambio inductivo.

• Asumir que hay una solución óptima que es diferente de la solución greedy. • Encontrar la "primera" diferencia entre las dos soluciones. • Argumentar que podemos intercambiar la elección óptima por la elección greedy sin empeorar la solución (aunque el intercambio podría no mejorarla).

Este argumento implica por inducción que alguna solución óptima contiene toda la solución greedy, y por lo tanto es igual a la solución greedy. A veces, como en el problema de programación, se requiere un paso adicional para mostrar que ninguna solución óptima mejora estrictamente la solución greedy.

**4.4 Códigos de Huffman**

Un código binario asigna una cadena de 0s y 1s a cada carácter en el alfabeto. Un código binario es libre de prefijos si ningún código es un prefijo de cualquier otro. (Confusamente, los códigos libres de prefijos también se llaman comúnmente códigos de prefijos.) ASCII de 7 bits y UTF-8 de Unicode son ambos códigos binarios libres de prefijos. El código Morse es un código binario con símbolos • y —, pero no es libre de prefijos, porque el código para E (•) es un prefijo de los códigos para I (••), S (•••), y H (••••).³

Cualquier código binario libre de prefijos puede visualizarse como un árbol binario con los caracteres codificados almacenados en las hojas. La palabra código para cualquier símbolo está dada por el camino desde la raíz hasta la hoja correspondiente; 0 para izquierda, 1 para derecha. Por lo tanto, la longitud de la palabra código de cualquier símbolo es la profundidad de la hoja correspondiente en el árbol de código. Aunque son superficialmente similares, los árboles de código binario no son árboles de búsqueda binaria; no nos importa en absoluto el orden de los símbolos en las hojas.

Supón que queremos codificar un mensaje escrito en un alfabeto de n caracteres para que el mensaje codificado sea lo más corto posible. Específicamente, dado un arreglo de conteos de frecuencia f[1 .. n], queremos calcular un código binario libre de prefijos que minimice la longitud total codificada del mensaje:

Σᵢ₌₁ⁿ f[i] · profundidad(i).

Esta es exactamente la misma función de costo que consideramos para optimizar árboles de búsqueda binaria, pero el problema de optimización es diferente, porque los árboles de código no están requeridos a mantener las claves en ningún orden particular.

³ Por esta razón, el código Morse puede describirse mejor como un código ternario libre de prefijos, con tres símbolos: •, —, y pausa. Alternativamente, el código Morse puede considerarse un código binario libre de prefijos, con un tiempo de sonido/luz/corriente/alto voltaje/humo/gas () y un tiempo de silencio/oscuridad/tierra/bajo voltaje/aire/líquido () como los dos símbolos. Entonces cada "dit" se codifica como , cada "dah" como , y cada pausa como . En el código Morse estándar, cada letra va seguida de una pausa, y cada palabra va seguida de dos pausas adicionales; sin embargo, las al final de todo el mensaje codificado se omiten. Por ejemplo, la cadena "MORSE CODE" se codifica sin ambigüedad como la siguiente cadena de bits: .

En 1951, como estudiante de doctorado en MIT, David Huffman desarrolló el siguiente algoritmo greedy para producir tal código óptimo:⁴

**Huffman:** Fusionar las dos letras menos frecuentes y recurrir.

El algoritmo de Huffman se ilustra mejor a través de un ejemplo. Supón que queremos codificar la siguiente oración útilmente auto-descriptiva, descubierta por Lee Sallows:⁵

*Esta oración contiene tres a's, tres c's, dos d's, veintiséis e's, cinco f's, tres g's, ocho h's, trece i's, dos l's, dieciséis n's, nueve o's, seis r's, veintisiete s's, veintidós t's, dos u's, cinco v's, ocho w's, cuatro x's, cinco y's, y solo una z.*

Para mantener las cosas simples, ignoremos los cuarenta y cuatro espacios, diecinueve apóstrofes, diecinueve comas, tres guiones, y solo un punto, y codifiquemos solo las letras, como si el mensaje estuviera escrito en scriptio continua:

THISSENTENCECONTAINSTHREEASTHREECSTWODSTWENTYSIXESFIVEFST HREEGSEIGHTHSTHIRTEENISTWOLSSIXTEENNSNINEOSSIXRSTWENTYSEV ENSSTWENTYTWOTSTWOUSFIVEVSEIGHTWSFOURXSFIVEYSANDONLYONEZ⁶

Aquí está la tabla de frecuencias para la oración de Sallows:

| **A** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **L** | **N** | **O** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 3 | 2 | 26 | 5 | 3 | 8 | 13 | 2 | 16 | 9 | 6 | 27 | 22 | 2 | 5 | 8 | 4 | 5 | 1 |

El algoritmo de Huffman escoge las dos letras menos frecuentes, rompiendo empates arbitrariamente—en este caso, digamos, Z y D—y las fusiona en un solo carácter nuevo DZ con frecuencia 3. Este nuevo carácter se convierte en un nodo interno en el árbol de código que estamos construyendo, con Z y D como sus hijos; no importa cuál hijo es cuál. El algoritmo entonces construye recursivamente un código de Huffman para la nueva tabla de frecuencias:

| **A** | **C** | **E** | **F** | **G** | **H** | **I** | **L** | **N** | **O** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **DZ** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 3 | 26 | 5 | 3 | 8 | 13 | 2 | 16 | 9 | 6 | 27 | 22 | 2 | 5 | 8 | 4 | 5 | 3 |

Después de 19 fusiones, todas las 20 letras han sido fusionadas. El registro de fusiones nos da nuestro árbol de código. El algoritmo hace un número de elecciones arbitrarias; como resultado, hay realmente varios códigos de Huffman diferentes. Un tal código de Huffman se muestra abajo; los números en nodos no-hoja son frecuencias para caracteres fusionados. Por ejemplo, el código para A es 101000, y el código para S es 111.

**[Referencia a Figura del árbol de Huffman]**

Codificar la oración de Sallows con este código de Huffman particular produciría una cadena de bits que comienza así:

100 0110 1011 111 111 110 010 100 110 010 101001 110 101001 0001 010 100 ··· T H I S S E N T E N C E C O N T

Aquí está la lista de costos para codificar cada carácter en la oración de Sallows, junto con la contribución de ese carácter a la longitud total de la oración codificada:

| **carácter** | **A** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **L** | **N** | **O** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| freq | 3 | 3 | 2 | 26 | 5 | 3 | 8 | 13 | 2 | 16 | 9 | 6 | 27 | 22 | 2 | 5 | 8 | 4 | 5 | 1 |
| profundidad | 6 | 6 | 7 | 3 | 5 | 6 | 4 | 4 | 6 | 3 | 4 | 5 | 3 | 3 | 6 | 5 | 4 | 5 | 5 | 7 |
| total | 18 | 18 | 14 | 78 | 25 | 18 | 32 | 52 | 12 | 48 | 36 | 30 | 81 | 66 | 12 | 25 | 32 | 20 | 25 | 7 |

En total, el mensaje codificado tiene 649 bits de longitud. Diferentes códigos de Huffman codifican los mismos caracteres de manera diferente, posiblemente con palabras código de longitud diferente, pero la longitud total del mensaje codificado es la misma para cada código de Huffman: 649 bits.

Dada la estructura simple del algoritmo de Huffman, es bastante sorprendente que produzca un código binario libre de prefijos óptimo.⁷ ¡Codificar la oración de Sallows usando cualquier código libre de prefijos requiere al menos 649 bits! Afortunadamente, la estructura recursiva hace que esta afirmación sea fácil de probar usando un argumento de intercambio, similar a nuestras demostraciones de optimalidad anteriores. Comenzamos probando que la primera elección del algoritmo es correcta.

**Lema 4.5.** Sean x e y los dos caracteres menos frecuentes (rompiendo empates entre caracteres igualmente frecuentes arbitrariamente). Hay un árbol de código óptimo en el que x e y son hermanos.

**Demostración:** Realmente probaré una declaración más fuerte: Hay un código óptimo en el que x e y son hermanos y tienen la mayor profundidad de cualquier hoja.

Sea T un árbol de código óptimo, y supón que este árbol tiene profundidad d. Porque T es un árbol binario completo, tiene al menos dos hojas a profundidad d que son hermanas. (¡Verifica esta afirmación por inducción!) Supón que esas dos hojas no son x e y, sino algunos otros caracteres a y b.

Sea T' el árbol de código obtenido intercambiando x y a, y sea Δ = d − profundidadₜ(x). Este intercambio aumenta la profundidad de x por Δ y disminuye la profundidad de a por Δ, así que

costo(T') = costo(T) + Δ · (f[x] − f[a]).

Nuestra suposición de que x es uno de los dos caracteres menos frecuentes pero a no lo es implica f[x] ≤ f[a], y nuestra suposición de que a tiene profundidad máxima implica Δ ≥ 0. Se sigue que costo(T') ≤ costo(T). Por otro lado, T es un árbol de código óptimo, así que también debemos tener costo(T') ≥ costo(T). Concluimos que T' también es un árbol de código óptimo.

Similarmente, intercambiar y y b debe dar otro árbol de código óptimo. En este árbol de código óptimo final, x e y son hermanos de profundidad máxima, como se requiere. □

Ahora la optimalidad está garantizada por nuestro querido amigo ¡el Hada de la Recursión! Nuestro argumento recursivo se basa en la siguiente definición recursiva no estándar: Un árbol binario completo es ya sea un solo nodo, o un árbol binario completo donde alguna hoja ha sido reemplazada por un nodo interno con dos hijos hoja.

**Teorema 4.6.** Cada código de Huffman es un código binario libre de prefijos óptimo.

**Demostración:** Si el mensaje tiene solo uno o dos caracteres distintos, el teorema es trivial, así que asume lo contrario.

Sean f[1 .. n] las frecuencias de entrada originales, y asume sin pérdida de generalidad que f[1] y f[2] son las dos frecuencias más pequeñas. Para establecer el subproblema recursivo, define f[n + 1] = f[1] + f[2]. Nuestro argumento de intercambio anterior implica que 1 y 2 son hermanos (más profundos) en algún código óptimo para f[1 .. n].

Sea T' el árbol de Huffman para f[3 .. n+1]; la hipótesis inductiva implica que T' es un árbol de código óptimo para el conjunto más pequeño de frecuencias. Para obtener el árbol de código final T, reemplazamos la hoja etiquetada n + 1 con un nodo interno con dos hijos, etiquetados 1 y 2. Afirmo que T es óptimo para el arreglo de frecuencias original f[1 .. n].

Para probar esta afirmación, podemos expresar el costo de T en términos del costo de T' como sigue. (En estas ecuaciones, profundidad(i) denota la profundidad de la hoja etiquetada i en ya sea T o T'; cada hoja que aparece en ambos T y T' tiene la misma profundidad en ambos árboles.)

costo(T) = Σᵢ₌₁ⁿ f[i] · profundidad(i) = Σᵢ₌₃ⁿ⁺¹ f[i] · profundidad(i) + f[1] · profundidad(1) + f[2] · profundidad(2) − f[n + 1] · profundidad(n + 1) = costo(T') + (f[1] + f[2]) · profundidadₜ − f[n + 1] · (profundidadₜ − 1) = costo(T') + f[1] + f[2] + (f[1] + f[2] − f[n + 1]) · (profundidadₜ − 1) = costo(T') + f[1] + f[2]

Esta ecuación implica que minimizar el costo de T es equivalente a minimizar el costo de T'; en particular, adjuntar hojas etiquetadas 1 y 2 a la hoja en T' etiquetada n + 1 da un árbol de código óptimo para las frecuencias originales. □

Para construir eficientemente un código de Huffman, mantenemos los caracteres en una cola de prioridad, usando las frecuencias de caracteres como prioridades. Podemos representar el árbol de código como tres arreglos de índices, listando los hijos Izquierdo y Derecho y el Padre de cada nodo. Las hojas del árbol de código final son nodos en los índices 1 hasta n, y la raíz es el nodo con índice 2n − 1. El pseudocódigo para el algoritmo se muestra en la Figura 4.4. BuildHuffman realiza O(n) operaciones de cola de prioridad: exactamente 2n − 1 Inserts y 2n − 2 ExtractMins. Si implementamos la cola de prioridad como un heap binario estándar, cada una de estas operaciones requiere tiempo O(log n), y por lo tanto todo el algoritmo ejecuta en tiempo O(n log n).

**BuildHuffman(f[1 .. n]):**

for i ← 1 to n

L[i] ← 0; R[i] ← 0

Insert(i, f[i])

for i ← n to 2n − 1

x ← ExtractMin() ⟨⟨encuentra dos símbolos más raros⟩⟩

y ← ExtractMin()

f[i] ← f[x] + f[y] ⟨⟨fusiona en un nuevo símbolo⟩⟩

Insert(i, f[i])

L[i] ← x; P[x] ← i ⟨⟨actualiza punteros del árbol⟩⟩

R[i] ← y; P[y] ← i

P[2n − 1] ← 0

**Figura 4.4.** Construyendo un código de Huffman.

Finalmente, algoritmos simples para codificar y decodificar mensajes usando un código de Huffman fijo se muestran en la Figura 4.5; ambos algoritmos ejecutan en tiempo O(m), donde m es la longitud del mensaje codificado.

**HuffmanEncode(A[1 .. k]):**

m ← 1

for i ← 1 to k

HuffmanEncodeOne(A[i])

HuffmanEncodeOne(x):

if x < 2n − 1

HuffmanEncodeOne(P[x])

if x = L[P[x]]

B[m] ← 0

else

B[m] ← 1

m ← m + 1

**HuffmanDecode(B[1 .. m]):**

k ← 1

v ← 2n − 1

for i ← 1 to m

if B[i] = 0

v ← L[v]

else

v ← R[v]

if L[v] = 0

A[k] ← v

k ← k + 1

v ← 2n − 1

**Figura 4.5.** Algoritmos de codificación y decodificación para códigos de Huffman

**4.5 Emparejamiento Estable**

Cada año, miles de nuevos doctores deben obtener internados en hospitales alrededor de Estados Unidos. Durante la primera mitad del siglo XX, la competencia entre hospitales por los mejores doctores llevó a ofertas cada vez más tempranas de internados, a veces tan temprano como el segundo año de la escuela de medicina, junto con plazos más estrictos para la aceptación. En los 1940s, las escuelas de medicina acordaron no liberar información hasta una fecha común durante el cuarto año de sus estudiantes. En respuesta, los hospitales comenzaron a exigir decisiones más rápidas. Para 1950, los hospitales regularmente llamarían a doctores, les ofrecerían internados, y exigirían respuestas inmediatas. Los internos fueron forzados a apostar si su hospital de tercera opción llamaba primero—aceptar y arriesgar perder una mejor oportunidad más tarde, o rechazar y arriesgar no tener posición alguna.⁸

⁸ El mercado laboral académico estadounidense involucra apuestas similares, al menos en ciencias de la computación. Algunos departamentos comienzan a hacer ofertas en febrero con plazos de decisión de dos semanas; otros departamentos ni siquiera comienzan a entrevistar hasta marzo; MIT notoriamente espera hasta mayo, cuando todas las entrevistas han terminado, antes de hacer cualquier oferta de facultad. Innecesario decir, la mezcla de fechas de ofertas y plazos de decisión causa tremendo estrés, tanto para candidatos como para departamentos. Por razones similares, desde 1965, la mayoría de las universidades estadounidenses han acordado una fecha límite común del 15 de abril para que estudiantes graduados prospectivos acepten ofertas de apoyo financiero (y por extensión, ofertas de admisión).

Finalmente, una cámara de compensación central para asignaciones de internado, ahora llamada el Programa Nacional de Emparejamiento de Residentes (NRMP), se estableció a principios de los 1950s. Cada año, los doctores envían una lista clasificada de todos los hospitales donde aceptarían un internado, y cada hospital envía una lista clasificada de doctores que aceptarían como internos. El NRMP entonces calcula un emparejamiento entre doctores y hospitales que satisface el siguiente requisito de estabilidad. Un emparejamiento es inestable si hay un doctor α y hospital B que estarían ambos más felices el uno con el otro que con su emparejamiento actual; es decir,

• α está emparejado con algún otro hospital A, aunque ella prefiere B. • B está emparejado con algún otro doctor β, aunque ellos prefieren α.

En este caso, llamamos (α, B) un par inestable para el emparejamiento. El objetivo del Emparejamiento de Residentes es un emparejamiento estable, que es un emparejamiento sin pares inestables.

Para simplicidad, asumiré de ahora en adelante que hay exactamente el mismo número de doctores y hospitales; cada hospital ofrece exactamente un internado; cada doctor clasifica todos los hospitales y viceversa; y finalmente, no hay empates en las clasificaciones de los doctores o hospitales.⁹

**Algunas Ideas Malas**

A primera vista, ¡ni siquiera está claro que un emparejamiento estable siempre exista! Ciertamente no todo emparejamiento de doctores y hospitales es estable. Supón que hay tres doctores (Dr. Quincy, Dr. Rotwang, Dr. Shephard, representados por letras minúsculas) y tres hospitales (Arkham Asylum, Bethlem Royal Hospital, y County General Hospital, representados por letras mayúsculas), que se clasifican entre sí como sigue:

q r s A B C

A C A r s q

C A B q q r

B B C s r s

El emparejamiento {Aq, Br, Cs} es inestable, porque Arkham preferiría contratar al Dr. Rotwang que al Dr. Quincy, y Dr. Rotwang preferiría trabajar en Arkham que en Bedlam. (A,r) es un par inestable para este emparejamiento.

Uno podría imaginar usar un algoritmo incremental que comience con un emparejamiento arbitrario, y luego greedily realice intercambios para resolver inestabilidades. Desafortunadamente, resolver una inestabilidad puede crear nuevas; de hecho, esta "mejora" incremental puede llevar a un bucle infinito. Por ejemplo, si comenzamos con nuestro emparejamiento inestable anterior {Aq, Br, Cs}, cada uno de los siguientes intercambios resuelve un par inestable (indicado sobre la flecha), pero la secuencia de intercambios lleva de vuelta al emparejamiento original:¹⁰

{Aq, Br, Cs} --Ar--> {Ar, Bq, Cs} --Cr--> {As, Bq, Cr} --Cq--> {As, Br, Cq} --Aq--> {Aq, Br, Cs}

Alternativamente, podríamos probar el siguiente protocolo greedy multi-ronda. En cada ronda, cada hospital no emparejado hace una oferta a su doctor favorito no emparejado, luego cada doctor no emparejado con una oferta acepta su oferta favorita. No es difícil probar que al menos un nuevo par doctor-hospital se empareja en cada ronda, así que el algoritmo siempre termina con un emparejamiento. Para el ejemplo de entrada anterior, ¡ya tenemos un emparejamiento estable {Ar, Bs, Cq} al final de la primera ronda! Pero considera la siguiente entrada en su lugar:

q r s A B C

C A A q q s

B C B s r r

A B C r s q

En la primera ronda, Dr. Shephard acepta una oferta de County, y Dr. Quincy acepta una oferta de Bedlam (rechazando la oferta de Arkham), dejando solo Dr. Rotwang y Arkham sin emparejar. Por lo tanto, el protocolo termina con el emparejamiento {Ar, Bq, Cs} después de dos rondas. Desafortunadamente, este emparejamiento es inestable; Arkham y Dr. Shephard se prefieren mutuamente a sus emparejamientos.

**Los Algoritmos Boston Pool y Gale-Shapley**

En 1952, el NRMP adoptó el algoritmo "Boston Pool" para asignar internos, así llamado porque había sido usado previamente por una cámara de compensación regional en el área de Boston. Diez años después, David Gale y Lloyd Shapley describieron y analizaron formalmente una generalización del algoritmo Boston Pool y probaron que calcula un emparejamiento estable. Gale y Shapley usaron la metáfora de admisiones universitarias. Esencialmente el mismo algoritmo fue desarrollado independientemente por Elliott Peranson en 1972 para uso en admisiones de escuelas de medicina. Algoritmos similares han sido adoptados desde entonces para muchos otros mercados de emparejamiento, incluyendo contratación de facultad en Francia, contratación de PhDs en economía nuevos en Estados Unidos, admisión universitaria en Alemania, admisión de escuelas públicas en Nueva York y Boston, asignaciones de camarotes para marineros de la Marina de EE.UU., y programas de emparejamiento de riñones.

Shapley fue galardonado con el Premio Nobel de Economía 2012 por su investigación sobre emparejamientos estables, junto con Alvin Roth, quien extendió significativamente el trabajo de Shapley y lo usó para desarrollar varios intercambios del mundo real. (Gale no compartió el premio, porque murió en 2008.)

Como nuestro último algoritmo greedy fallido, el algoritmo Gale-Shapley procede en rondas hasta que cada posición ha sido aceptada. Cada ronda tiene dos etapas:

1. Un hospital A no emparejado arbitrario ofrece su posición al mejor doctor α (según la lista de preferencias de A) que no lo ha rechazado ya.
2. Si α no está emparejado, ella (tentativamente) acepta la oferta de A. Si α ya tiene un emparejamiento pero prefiere A, ella rechaza su emparejamiento actual y (tentativamente) acepta la nueva oferta de A. De lo contrario, α rechaza la nueva oferta.

Cada doctor finalmente acepta la mejor oferta que recibe, según su lista de preferencias.¹¹ En resumen, los hospitales hacen ofertas greedily, y los doctores aceptan ofertas greedily. La capacidad de los doctores de rechazar sus emparejamientos actuales en favor de mejores ofertas es la clave para hacer que esta estrategia greedy mutua funcione.

Por ejemplo, supón que hay cuatro doctores (Dr. Quincy, Dr. Rotwang, Dr. Shephard, y Dr. Tam) y cuatro hospitales (Arkham Asylum, Bethlem Royal Hospital, County General Hospital, y The Dharma Initiative), que se clasifican entre sí como sigue:

q r s t A B C D

A A B D t r t s

B D A B s t r r

C C C C r q s q

D B D A q s q t

Dadas estas listas de preferencias como entrada, el algoritmo Gale-Shapley podría proceder como sigue:

1. Arkham hace una oferta a Dr. Tam.
2. Bedlam hace una oferta a Dr. Rotwang.
3. County hace una oferta a Dr. Tam, quien rechaza su oferta anterior de Arkham.
4. Dharma hace una oferta a Dr. Shephard. (De este punto en adelante, hay solo un hospital no emparejado, así que el algoritmo no tiene más elecciones.)
5. Arkham hace una oferta a Dr. Shephard, quien rechaza su oferta anterior de Dharma.
6. Dharma hace una oferta a Dr. Rotwang, quien rechaza su oferta anterior de Bedlam.
7. Bedlam hace una oferta a Dr. Tam, quien rechaza su oferta anterior de County.
8. County hace una oferta a Dr. Rotwang, quien la rechaza.
9. County hace una oferta a Dr. Shephard, quien la rechaza.
10. County hace una oferta a Dr. Quincy.

Después de la décima ronda, todas las ofertas pendientes son aceptadas, y el algoritmo devuelve el emparejamiento {As, Bt, Cq, Dr}. Puedes (y debes) verificar por fuerza bruta que este emparejamiento es estable, aunque ningún doctor fue contratado por su hospital favorito, y ningún hospital contrató a su doctor favorito; de hecho, County terminó contratando a su doctor menos favorito. Este no es el único emparejamiento estable para estas listas de preferencias; el emparejamiento {Ar, Bs, Cq, Dt} también es estable.

**Tiempo de Ejecución**

Analizar el número de ofertas realizadas por el algoritmo es relativamente directo (razón por la cual lo estamos haciendo primero). Cada hospital hace una oferta a cada doctor como máximo una vez, así que el algoritmo hace como máximo n² ofertas.

Para analizar el tiempo de ejecución real, sin embargo, necesitamos especificar el algoritmo en más detalle. ¿Cómo se dan las listas de preferencias al algoritmo? ¿Cómo decide el algoritmo si algún hospital está no emparejado, y si es así, cómo encuentra un hospital no emparejado? ¿Cómo almacena el algoritmo los emparejamientos tentativos? ¿Cómo decide el algoritmo si un doctor prefiere su nueva oferta a su emparejamiento actual? Más fundamentalmente: ¿Cómo representa realmente el algoritmo doctores y hospitales?

Una posibilidad es representar cada doctor y hospital por un entero único entre 1 y n, y representar preferencias como dos arreglos Dpref[1 .. n, 1 .. n] y Hpref[1 .. n, 1 .. n], donde Dpref[i,r] representa el r-ésimo hospital en la lista de preferencias del doctor i, y HPref[j,r] representa el r-ésimo doctor en la lista de preferencias del hospital j. Con la entrada en esta forma, el algoritmo Boston Pool puede ejecutar cada oferta en tiempo constante, después de algún preprocesamiento inicial; la implementación general ejecuta en tiempo O(n²). Dejamos los detalles restantes como un ejercicio directo.

Un ejercicio algo más difícil es probar que hay entradas (y elecciones de quién hace ofertas cuándo) que fuerzan Ω(n²) ofertas antes de que el algoritmo se detenga. Por lo tanto, nuestro límite superior O(n²) en el tiempo de ejecución del peor caso es ajustado.

**Corrección**

Pero ¿por qué es correcto el algoritmo en absoluto? ¿Cómo sabemos que siempre calcula un emparejamiento estable, o cualquier emparejamiento completo para el caso?

Una vez que un doctor recibe una oferta, ella tiene al menos un emparejamiento tentativo por el resto del tiempo. Equivalentemente, si algún doctor está no emparejado, entonces ningún hospital ha ofrecido a ese doctor un trabajo, lo que implica que los hospitales no han agotado sus listas de preferencias. Se sigue que cuando el algoritmo termina (después de como máximo n² rondas), cada doctor está emparejado, y por lo tanto cada posición está llena. En otras palabras, el algoritmo siempre calcula un emparejamiento perfecto entre doctores y hospitales. (¡Uf!) Solo queda probar que el emparejamiento resultante es estable.

Supón que el algoritmo empareja algún doctor α a algún hospital A, aunque ella prefiere otro hospital B. Porque cada doctor acepta la mejor oferta que recibe, α no recibió oferta que le gustara más que A; en particular, B nunca hizo una oferta a α. Por otro lado, B hizo ofertas a cada doctor que ellos prefieren sobre su emparejamiento final β. Se sigue que B prefiere β sobre α, lo que significa que (α, B) no es un par inestable. Concluimos que no hay pares inestables; ¡el emparejamiento es estable!

**¡Optimalidad!**

Sorprendentemente, la corrección del algoritmo Gale-Shapley no depende de qué hospital hace su oferta en cada ronda. De hecho, no importa qué hospital no asignado haga una oferta en cada ronda, ¡el algoritmo siempre calcula el mismo emparejamiento! Digamos que α es un doctor factible para A si hay un emparejamiento estable que asigna al doctor α al hospital A.

**Lema 4.7.** Durante el algoritmo Gale-Shapley, cada hospital A es rechazado solo por doctores que son no factibles para A.

**Demostración:** Probamos el lema por inducción en el número de rondas. Considera una ronda arbitraria del algoritmo, en la cual el doctor α rechaza un hospital A por otro hospital B. El rechazo implica que α prefiere B a A. Cada doctor que aparece más alto que α en la lista de preferencias de B ya rechazó a B en una ronda anterior y por lo tanto, por la hipótesis inductiva, es no factible para B.

Ahora considera un emparejamiento arbitrario (de los mismos doctores y hospitales) que asigna α a A. Ya establecimos que α prefiere B a A. Si B prefiere α a su pareja, el emparejamiento es inestable. Por otro lado, si B prefiere su pareja a α, entonces (por nuestro argumento anterior) su pareja es no factible, y nuevamente el emparejamiento es inestable. Concluimos que no hay emparejamiento estable que asigne α a A. □

Ahora sea mejor(A) el doctor clasificado más alto factible en la lista de preferencias de A. El Lema 4.7 implica que cada doctor que A prefiere a su emparejamiento final es no factible para A. Por otro lado, el emparejamiento final es estable, así que el doctor asignado a A debe ser factible para A. El siguiente resultado es ahora inmediato:

**Corolario 4.8.** El algoritmo Gale-Shapley empareja mejor(A) con A, para cada hospital A.

En otras palabras, el algoritmo Gale-Shapley calcula el mejor emparejamiento estable posible desde el punto de vista de los hospitales. ¡Resulta que este emparejamiento también es el peor posible desde el punto de vista de los doctores! Sea peor(α) el hospital clasificado más bajo factible en la lista de preferencias del doctor α.

**Corolario 4.9.** El algoritmo Gale-Shapley empareja α con peor(α), para cada doctor α.

**Demostración:** Supón que Gale y Shapley asignan al doctor α al hospital A; necesitamos mostrar que A = peor(α). Considera un emparejamiento estable arbitrario donde A no está emparejado con α sino con otro doctor β. El corolario anterior implica que A prefiere α = mejor(A) a β. Porque el emparejamiento es estable, α debe por lo tanto preferir su hospital asignado a A. Este argumento funciona para cualquier emparejamiento estable, así que α prefiere cada otro emparejamiento factible a A; en otras palabras, A = peor(α). □

Una consecuencia sutil de estos dos corolarios, descubierta por Lester Dubins y David Freedman en 1981, es que un doctor puede potencialmente mejorar su emparejamiento mintiendo sobre sus preferencias, pero un hospital no puede. (Sin embargo, un conjunto de hospitales puede conspirar para que algunos de sus emparejamientos mejoren.) Parcialmente por esta razón, el Programa Nacional de Emparejamiento de Residencias revirtió su algoritmo de emparejamiento en 1998, para que residentes potenciales ofrezcan trabajar para hospitales, según sus órdenes de preferencia, y cada hospital acepta su mejor oferta. Por lo tanto, el nuevo algoritmo calcula el mejor emparejamiento estable posible para los doctores, y el peor emparejamiento estable posible para los hospitales. En la práctica, sin embargo, esta reversión alteró menos del 1% de los emparejamientos de los residentes. Hasta donde sé, el efecto preciso de este cambio en los pacientes es un problema abierto.

**Ejercicios**

**Caveat lector:** ¡Algunos de estos ejercicios no pueden resolverse usando algoritmos greedy! Siempre que describas y analices un algoritmo greedy, también debes incluir una demostración de que tu algoritmo es correcto; esta demostración típicamente tomará la forma de un argumento de intercambio. Estas demostraciones son especialmente importantes en clases (como la mía) que normalmente no requieren demostraciones de corrección.

1. El algoritmo GreedySchedule que describimos para el problema de programación de clases no es la única estrategia greedy que podríamos haber probado. Para cada una de las siguientes estrategias greedy alternativas, ya sea prueba que el algoritmo resultante siempre construye un horario óptimo, o describe un pequeño ejemplo de entrada para el cual el algoritmo no produce un horario óptimo. Asume que todos los algoritmos rompen empates arbitrariamente (es decir, de una manera que está completamente fuera de tu control). [Pista: ¡Tres de estos algoritmos son realmente correctos.]

(a) Elige el curso x que termina último, descarta clases que conflicten con x, y recurre. (b) Elige el curso x que comienza primero, descarta todas las clases que conflicten con x, y recurre. (c) Elige el curso x que comienza último, descarta todas las clases que conflicten con x, y recurre. (d) Elige el curso x con duración más corta, descarta todas las clases que conflicten con x, y recurre. (e) Elige un curso x que conflicte con el menor número de otros cursos, descarta todas las clases que conflicten con x, y recurre. (f) Si ninguna clase conflicte, elígelas todas. De lo contrario, descarta el curso con duración más larga y recurre. (g) Si ninguna clase conflicte, elígelas todas. De lo contrario, descarta un curso que conflicte con la mayoría de otros cursos y recurre. (h) Sea x la clase con el tiempo de inicio más temprano, y sea y la clase con el segundo tiempo de inicio más temprano. • Si x e y son disjuntos, elige x y recurre en todo excepto x. • Si x contiene completamente a y, descarta x y recurre. • De lo contrario, descarta y y recurre. (i) Si algún curso x contiene completamente otro curso, descarta x y recurre. De lo contrario, elige el curso y que termina último, descarta todas las clases que conflicten con y, y recurre.

1. Ahora considera una versión ponderada del problema de programación de clases, donde diferentes clases ofrecen diferentes números de horas de crédito (totalmente no relacionadas con la duración de las conferencias de la clase). Tu objetivo ahora es elegir un conjunto de clases que no conflicten que te den el mayor número posible de horas de crédito, dados arreglos de tiempos de inicio, tiempos de finalización, y horas de crédito como entrada.

(a) Prueba que el algoritmo greedy descrito al principio de este capítulo—Elige la clase que termina primero y recurre—no siempre devuelve un horario óptimo. (b) Prueba que ninguno de los algoritmos greedy descritos en el Ejercicio 1 siempre devuelve un horario óptimo. [Pista: Resuelve el Ejercicio 1 primero; los algoritmos que no funcionan allí tampoco funcionan aquí.] (c) Describe y analiza un algoritmo que siempre calcula un horario óptimo. [Pista: Tu algoritmo no será greedy.]

1. Sea X un conjunto de n intervalos en la recta real. Decimos que un subconjunto de intervalos Y ⊆ X cubre X si la unión de todos los intervalos en Y es igual a la unión de todos los intervalos en X. El tamaño de una cubierta es simplemente el número de intervalos.

Describe y analiza un algoritmo eficiente para calcular la cubierta más pequeña de X. Asume que tu entrada consiste en dos arreglos L[1 .. n] y R[1 .. n], representando los puntos finales izquierdo y derecho de los intervalos en X. Si usas un algoritmo greedy, debes probar que es correcto.

**[Referencia a Figura]** Un conjunto de intervalos, con una cubierta (sombreada) de tamaño 7.

1. Sea X un conjunto de n intervalos en la recta real. Decimos que un conjunto P de puntos apuñala X si cada intervalo en X contiene al menos un punto en P. Describe y analiza un algoritmo eficiente para calcular el conjunto más pequeño de puntos que apuñala X. Asume que tu entrada consiste en dos arreglos L[1 .. n] y R[1 .. n], representando los puntos finales izquierdo y derecho de los intervalos en X. Como es usual, si usas un algoritmo greedy, debes probar que es correcto.

**[Referencia a Figura]** Un conjunto de intervalos apuñalados por cuatro puntos (mostrados aquí como segmentos verticales)

1. Sea X un conjunto de n intervalos en la recta real. Una coloración propia de X asigna un color a cada intervalo, para que dos intervalos que se superpongan sean asignados colores diferentes. Describe y analiza un algoritmo eficiente para calcular el número mínimo de colores necesarios para colorear propiamente X. Asume que tu entrada consiste en dos arreglos L[1 .. n] y R[1 .. n], representando los puntos finales izquierdo y derecho de los intervalos en X. Como es usual, si usas un algoritmo greedy, debes probar que es correcto.

**[Referencia a Figura]** Una coloración propia de un conjunto de intervalos usando cinco colores.

1. (a) Para cada entero n, encuentra un arreglo de frecuencias f[1 .. n] cuyo árbol de código de Huffman tenga profundidad n − 1, tal que la frecuencia más grande sea lo más pequeña posible.

(b) Supón que la longitud total N del mensaje no codificado está acotada por un polinomio en el tamaño del alfabeto n. Prueba que cualquier árbol de Huffman para las frecuencias f[1 .. n] tiene profundidad O(log n).

♥7. Llama a un arreglo de frecuencias f[1 .. n] α-pesado si satisface dos condiciones: • f[1] > f[i] para todo i > 1; es decir, 1 es el símbolo más frecuente único. • f[1] ≥ α Σᵢ₌₁ⁿ f[i]; es decir, al menos una fracción α de los símbolos son 1s.

Encuentra el número real más grande α tal que en cada código de Huffman para cada arreglo de frecuencias α-pesado, el símbolo 1 está representado por un solo bit. [Pista: Primero prueba que 1/3 ≤ α ≤ 1/2.]

1. Describe y analiza un algoritmo para calcular un código ternario libre de prefijos óptimo para un arreglo dado de frecuencias f[1 .. n]. No olvides probar que tu algoritmo es correcto para todo n.
2. Describe en detalle cómo implementar el algoritmo de emparejamiento estable Gale-Shapley, para que el tiempo de ejecución del peor caso sea O(n²), como se afirmó anteriormente en este capítulo.
3. (a) Prueba que es posible que el algoritmo Gale-Shapley realice Ω(n²) ofertas antes de la terminación. (Necesitas describir tanto una entrada adecuada como una secuencia de Ω(n²) ofertas válidas.)

(b) Describe para cualquier entero n un conjunto de preferencias para n doctores y n hospitales que fuerce al algoritmo Gale-Shapley a ejecutar Ω(n²) rondas, sin importar qué propuesta válida se haga en cada ronda. [Pista: La parte (b) implica la parte (a).]

1. Describe y analiza un algoritmo eficiente para determinar si un conjunto dado de preferencias de hospitales y doctores tiene un emparejamiento estable único.
2. Considera una generalización del problema de emparejamiento estable, donde algunos doctores no clasifican todos los hospitales y algunos hospitales no clasifican todos los doctores, y un doctor puede ser asignado a un hospital solo si cada uno aparece en la lista de preferencias del otro. En este caso, hay tres situaciones inestables adicionales: • Un hospital emparejado prefiere un doctor no emparejado a su emparejamiento asignado. • Un doctor emparejado prefiere un hospital no emparejado a su emparejamiento asignado. • Un doctor no emparejado y un hospital no emparejado aparecen en las listas de preferencias del otro.

Un emparejamiento estable en este entorno puede dejar algunos doctores y/o hospitales no emparejados, aunque sus listas de preferencias no estén vacías. Por ejemplo, si cada doctor lista a Harvard como su único hospital aceptable, y cada hospital lista al Dr. House como su único interno aceptable, entonces solo House y Harvard serán emparejados.

Describe y analiza un algoritmo eficiente que calcule un emparejamiento estable en este entorno más general. [Pista: Reduce a una instancia donde cada doctor clasifica cada hospital y viceversa, y luego invoca Gale-Shapley.]

1. La empresa de muebles escandinava Fürni ha contratado n conductores para entregar n órdenes idénticas a n direcciones diferentes en Wilmington, Delaware. Cada conductor tiene su propia ruta de entrega bien establecida a través de Wilmington que visita todas las n direcciones. Asumiendo que siguen sus rutas como siempre lo hacen, dos conductores nunca visitan las mismas direcciones al mismo tiempo.

En principio, cada uno de los n conductores puede entregar sus muebles a cualquiera de las n direcciones, pero hay una complicación. Uno de los conductores ha conectado secretamente sensores de proximidad y explosivos a los sofás Johannshamn (con el patrón de rayas verdes Strinne). Si dos sofás están alguna vez en la misma dirección al mismo tiempo, ambos explotarán, destruyendo tanto el camión de entrega como el edificio en esa dirección. Esto solo puede pasar si un conductor entrega una orden a esa dirección, y luego otro conductor visita esa misma dirección mientras los muebles aún están en su camión.

Tu trabajo como despachador de Fürni es asignar cada conductor a una dirección de entrega. Describe un algoritmo para asignar direcciones a conductores para que cada una de las n direcciones reciba su orden de muebles y no haya explosiones.

Por ejemplo, supón que la ruta de Jack visita 537 Paper Street a las 6pm y 1888 Franklin Street a las 8pm, y la ruta de Marla visita 537 Paper a las 7pm y 1888 Franklin a las 9pm. Entonces Jack debería entregar a 1888 Franklin, y Marla debería entregar a 537 Paper; de lo contrario, habría una explosión en 1888 Franklin a las 8pm. (Cue the Pixies.) [Pista: Jack y Marla son un poco inestables.]

1. Supón que eres un simple tendero viviendo en un país con n tipos diferentes de monedas, con valores 1 = c[1] < c[2] < ··· < c[n]. (En los EE.UU., por ejemplo, n = 6 y los valores son 1, 5, 10, 25, 50 y 100 centavos.) Tu querido y benevolente dictador, El Generalísimo, ha decretado que cuando des cambio a un cliente, debes usar el menor número posible de monedas, para no desgastar la imagen de El Generalísimo cariñosamente grabada en cada moneda por sirvientes del Tesoro Real.

(a) En Estados Unidos, hay un algoritmo greedy simple que siempre resulta en el menor número de monedas: resta la moneda más grande y recursivamente da cambio por el resto. El Generalísimo no aprueba la codicia capitalista americana. Muestra que hay un conjunto de valores de monedas para el cual el algoritmo greedy no siempre da el menor número posible de monedas.

(b) Ahora supón que El Generalísimo decide imponer un sistema monetario donde las denominaciones de monedas son potencias consecutivas b⁰, b¹, b², . . . , bᵏ de algún entero b ≥ 2. Prueba que a pesar de la desaprobación de El Generalísimo, el algoritmo greedy descrito en la parte (a) sí hace cambio óptimo en este sistema monetario.

(c) Describe y analiza un algoritmo eficiente para determinar, dado un monto objetivo T y un arreglo ordenado c[1 .. n] de denominaciones de monedas, el menor número de monedas necesarias para hacer T centavos en cambio. Asume que c[1] = 1, para que sea posible hacer cambio por cualquier monto T.

1. Supón que se te da un arreglo A[1 .. n] de enteros, cada uno de los cuales puede ser positivo, negativo, o cero. Un subarreglo contiguo A[i .. j] se llama un intervalo positivo si la suma de sus entradas es mayor que cero. Describe y analiza un algoritmo para calcular el número mínimo de intervalos positivos que cubren cada entrada positiva en A. Por ejemplo, dado el siguiente arreglo como entrada, tu algoritmo debería devolver 3. Si cada entrada en el arreglo de entrada es negativa, tu algoritmo debería devolver 0.

**[Referencia a ejemplo con sumas]**

1. Considera el siguiente proceso. En todo momento tienes un solo entero positivo x, que inicialmente es igual a 1. En cada paso, puedes ya sea incrementar x o duplicar x. Tu objetivo es producir un valor objetivo n. Por ejemplo, puedes producir el entero 10 en cuatro pasos como sigue:

1 --+1--> 2 --×2--> 4 --+1--> 5 --×2--> 10

Obviamente puedes producir cualquier entero n usando exactamente n − 1 incrementos, pero para casi todos los valores de n, esto es horriblemente ineficiente. Describe y analiza un algoritmo para calcular el número mínimo de pasos requeridos para producir cualquier entero dado n.

1. Supón que tenemos n esquiadores con alturas dadas en un arreglo P[1 .. n], y n esquís con alturas dadas en un arreglo S[1 .. n]. Describe un algoritmo eficiente para asignar un esquí a cada esquiador, para que la diferencia promedio entre la altura de un esquiador y su esquí asignado sea lo más pequeña posible. El algoritmo debería calcular una permutación σ tal que la expresión

(1/n) Σᵢ₌₁ⁿ |P[i] − S[σ(i)]|

sea lo más pequeña posible.

1. Alice quiere dar una fiesta y está tratando de decidir a quién invitar. Tiene n personas para elegir, y sabe qué pares de estas personas se conocen entre sí. Quiere invitar a tantas personas como sea posible, sujeto a dos restricciones: • Para cada invitado, debería haber al menos cinco otros invitados que ya conocen. • Para cada invitado, debería haber al menos cinco otros invitados que no conocen ya.

Describe y analiza un algoritmo que calcule el mayor número posible de invitados que Alice puede invitar, dada una lista de n personas y la lista de pares que se conocen entre sí.

1. Supón que se nos dan dos arreglos C[1 .. n] y R[1 .. n] de enteros positivos. Una matriz n × n de 0s y 1s concuerda con R y C si, para cada índice i, la i-ésima fila contiene R[i] 1s, y la i-ésima columna contiene C[i] 1s. Describe y analiza un algoritmo que ya sea construya una matriz que concuerde con R y C, o reporte correctamente que no existe tal matriz.
2. Acabas de aceptar un trabajo de Elon Musk, entregando burritos de San Francisco a la Ciudad de Nueva York. Puedes manejar un Vehículo de Entrega de Burritos a través del nuevo Tubo Transcontinental Subterráneo de Entrega de Burritos de Elon, que corre en línea directa entre estas dos ciudades.¹²

Tu Vehículo de Entrega de Burritos funciona con baterías de un solo uso, que deben ser reemplazadas después de como máximo 100 millas. El combustible real es virtualmente gratis, pero las baterías son caras y frágiles, y por lo tanto deben ser instaladas solo por miembros oficiales del Sindicato de Técnicos de Reemplazo de Baterías de Vehículos de Entrega de Burritos Subterráneos Transcontinentales.¹³ Por lo tanto, incluso si reemplazas tu batería temprano, aún debes pagar el precio completo por cada nueva batería a instalar. Además, tu Vehículo es demasiado pequeño para llevar más de una batería a la vez.

Hay varias estaciones de combustible a lo largo del Tubo; cada estación cobra un precio diferente por instalar una nueva batería. Antes de comenzar tu viaje, imprimes cuidadosamente la página de Wikipedia que lista las ubicaciones y precios de cada estación de combustible a lo largo del Tubo. Dada esta información, ¿cómo decides los mejores lugares para parar por combustible?

Más formalmente, supón que se te dan dos arreglos D[1 .. n] y C[1 .. n], donde D[i] es la distancia desde el inicio del Tubo hasta la i-ésima estación, y C[i] es el costo de reemplazar tu batería en la i-ésima estación. Asume que tu viaje comienza y termina en estaciones de combustible (así que D[1] = 0 y D[n] es la longitud total de tu viaje), y que tu auto comienza con una batería vacía (así que debes instalar una nueva batería en la estación 1).

(a) Describe y analiza un algoritmo greedy para encontrar el número mínimo de paradas de reabastecimiento necesarias para completar tu viaje. No olvides probar que tu algoritmo es correcto.

(b) Pero lo que realmente quieres minimizar es el costo total de viaje. Muestra que tu algoritmo greedy en la parte (a) no produce una solución óptima cuando se extiende a este entorno.

(c) Describe un algoritmo eficiente para calcular las ubicaciones de las estaciones de combustible en las que deberías parar para minimizar el costo total de viaje.

1. Te han contratado para almacenar una secuencia de n libros en estantes en una biblioteca. El orden de los libros está fijado por el sistema de catalogación y no puede cambiarse; cada estante debe almacenar un intervalo contiguo de la secuencia dada de libros. Se te dan dos arreglos H[1 .. n] y T[1 .. n], donde H[i] y T[i] son respectivamente la altura y el grosor del i-ésimo libro en la secuencia. Todos los estantes en esta biblioteca tienen la misma longitud L; el grosor total de todos los libros en cualquier estante individual no puede exceder L.

(a) Supón que todos los libros tienen la misma altura h y los estantes tienen altura mayor que h, así que cada libro cabe en cada estante. Describe y analiza un algoritmo greedy para almacenar los libros en el menor número de estantes posible. [Pista: El algoritmo es obvio, pero ¿por qué es correcto?]

(b) Ese fue un buen calentamiento, pero ahora aquí está el problema real. De hecho, los libros tienen alturas diferentes, pero puedes ajustar la altura de cada estante para que coincida con el libro más alto en ese estante. (En particular, puedes cambiar la altura de cualquier estante vacío a cero.) Ahora tu tarea es almacenar los libros para que la suma de las alturas de los estantes sea lo más pequeña posible. Muestra que tu algoritmo greedy de la parte (a) no siempre da la mejor solución a este problema.

(c) Describe y analiza un algoritmo para encontrar el mejor emparejamiento entre libros y estantes como se describe en la parte (b).

1. Una cadena w de paréntesis ( y ) está balanceada si satisface una de las siguientes condiciones: • w es la cadena vacía. • w = (x) para alguna cadena balanceada x • w = xy para algunas cadenas balanceadas x e y

Por ejemplo, la cadena w = ((())()())(()())() está balanceada, porque w = xy, donde x = ((())()()) e y = (()())().

(a) Describe y analiza un algoritmo para determinar si una cadena dada de paréntesis está balanceada.

(b) Describe y analiza un algoritmo greedy para calcular la longitud de una subsecuencia balanceada más larga de una cadena dada de paréntesis. Como es usual, no olvides probar que tu algoritmo es correcto.

Para ambos problemas, tu entrada es un arreglo w[1 .. n], donde para cada i, ya sea w[i] = ( o w[i] = ). Ambos de tus algoritmos deberían ejecutar en tiempo O(n).

1. Un día Alex se cansó de escalar en un gimnasio y decidió llevar a un gran grupo de amigos escaladores afuera a escalar. Fueron a un área de escalada con una gran roca ancha, no muy alta, con varios agarres marcados para manos y pies. Alex rápidamente determinó un conjunto "permitido" de movimientos que su grupo de amigos puede realizar para ir de un agarre a otro.

El sistema general de agarres puede describirse por un árbol arraigado T con n vértices, donde cada vértice corresponde a un agarre y cada arista corresponde a un movimiento permitido entre agarres. Los caminos de escalada convergen a medida que suben la roca, llevando a un agarre único en la cumbre, representado por la raíz de T.

Alex y sus amigos (que son todos escaladores excelentes) decidieron jugar un juego, donde tantos escaladores como sea posible están simultáneamente en la roca y cada escalador necesita realizar una secuencia de exactamente k movimientos. Cada escalador puede elegir un agarre arbitrario para comenzar, y todos los movimientos deben moverse alejándose del suelo. Por lo tanto, cada escalador traza un camino de k aristas en el árbol T, todas dirigidas hacia la raíz. Sin embargo, no se permite que dos escaladores toquen el mismo agarre; los caminos seguidos por escaladores diferentes no pueden intersecarse en absoluto.

(a) Describe y analiza un algoritmo greedy para calcular el número máximo de escaladores que pueden jugar este juego. Tu algoritmo recibe un árbol arraigado T y un entero k como entrada, y debería calcular el mayor número posible de caminos disjuntos en T, donde cada camino tiene longitud k. No asumas que T es un árbol binario. Por ejemplo, dado el árbol de abajo como entrada, tu algoritmo debería devolver el entero 8.

**Figura 4.6.** Siete caminos disjuntos de longitud k = 3. Este no es el conjunto más grande de tales caminos en este árbol.

(b) Ahora supón que cada vértice en T tiene una recompensa asociada, y tu objetivo es maximizar la recompensa total de los vértices en tus caminos, en lugar del número total de caminos. Muestra que tu algoritmo greedy no siempre devuelve la recompensa óptima.

(c) Describe un algoritmo eficiente para calcular la recompensa máxima posible, como se describe en la parte (b).

1. ¡Felicitaciones! Has conquistado exitosamente Camelot, transformando la antigua monarquía hereditaria llena de cicatrices de batalla en una comuna anarco-sindicalista, donde los ciudadanos toman turnos para actuar como una especie de oficial-ejecutivo-por-la-semana, pero con todas las decisiones de ese oficial ratificadas en una reunión especial bi-semanal, por una mayoría simple en el caso de asuntos puramente internos, pero por una mayoría de dos tercios en el caso de más mayor...

Como acto simbólico final, ordenas que la Mesa Redonda (sorprendentemente, una mesa circular real) sea dividida en cuñas tipo pizza y distribuida a los ciudadanos de Camelot como trofeos. Cada ciudadano ha enviado una solicitud para una cuña angular de la mesa, especificada por dos ángulos—por ejemplo: Sir Robin el Valiente podría solicitar la cuña desde 17.23° hasta 42°, y Sir Lancelot el Puro podría solicitar la cuña de 2° desde 359° hasta 1°. Cada ciudadano estará feliz si y solo si recibe precisamente la cuña que solicitó.

Desafortunadamente, algunos de estos rangos se superponen, así que satisfacer todas las solicitudes de los ciudadanos es simplemente imposible. Bienvenido a la política.

Describe y analiza un algoritmo para encontrar el número máximo de solicitudes que pueden ser satisfechas. [Pista: La salida de tu algoritmo no debería cambiar si rotas la mesa. No asumas que los ángulos son enteros.]

1. Supón que estás parado en un campo rodeado por varios globos grandes. Quieres usar tu nueva Acme Brand Zap-O-Matic™ para reventar todos los globos, sin moverte de tu ubicación actual. La Zap-O-Matic™ dispara un rayo láser de alta potencia, que revienta todos los globos que golpea. Dado que cada disparo requiere suficiente energía para alimentar un pequeño país por un año, quieres disparar el menor número de disparos posible.

**[Referencia a Figura 4.7]** Nueve globos reventados por cuatro disparos de la Zap-O-Matic™

El problema mínimo de zap puede plantearse más formalmente como sigue. Dado un conjunto C de n círculos en el plano, cada uno especificado por su radio y las coordenadas (x, y) de su centro, calcula el número mínimo de rayos desde el origen que intersectan cada círculo en C. Tu objetivo es encontrar un algoritmo eficiente para este problema.

(a) Supón que es posible disparar un rayo que no intersecte ningún globo. Describe y analiza un algoritmo greedy que resuelve el problema mínimo de zap en este caso especial. [Pista: Ve el Ejercicio 4.]

(b) Describe y analiza un algoritmo greedy cuya salida esté dentro de 1 del óptimo. Es decir, si m es el número mínimo de rayos requerido para golpear cada globo, entonces tu algoritmo greedy debe devolver ya sea m o m+1. (Por supuesto, debes probar este hecho.)

(c) Describe un algoritmo que resuelve el problema mínimo de zap en tiempo O(n²).

♥(d) Describe un algoritmo que resuelve el problema mínimo de zap en tiempo O(n log n).

Asume que tienes una subrutina que te dice el rango de ángulos de rayos que intersecta un círculo arbitrario c en tiempo O(1). Esta subrutina no es difícil de escribir, pero no es la parte interesante del problema.

**Notas al pie:**

¹ Los lectores que estén tentados a objetar que la cinta magnética ha sido obsoleta por décadas están cordialmente invitados a visitar su instalación de supercomputación más cercana; pregunta por ver los robots de cinta.

² Pero aún deberías trabajar los detalles tú mismo.

³ Por esta razón, el código Morse puede describirse mejor como un código ternario libre de prefijos.

⁴ Huffman era estudiante en una clase de teoría de la información enseñada por Robert Fano.

⁵ Esta oración fue reportada por primera vez por Alexander Dewdney en su columna "Computer Recreations" de octubre de 1984 en Scientific American.

⁶ ... y él habló por cuarenta y cinco minutos, y nadie entendió una palabra de lo que dijo.

⁷ Ciertamente fue sorprendente tanto para Huffman como para Fano.

⁸ El mercado laboral académico estadounidense involucra apuestas similares.

⁹ En realidad, la mayoría de los hospitales ofrecen múltiples internados.

¹⁰ Este ejemplo fue descubierto por Donald Knuth.

¹¹ El algoritmo Boston Pool de 1952 es un caso especial del algoritmo Gale-Shapley.

¹² ... y que claramente fue modelado según el ficticio "Alameda-Weehauken Burrito Tunnel" de Maciej Cegłowski.

¹³ o como se llaman a sí mismos en alemán, Die Transkontinentaluntergrundburritolieferfahrzeugbatteriewechseltechnikervereinigung.